92个实例、1个阶段性案例、1个通用模块、3个项目实战案例
资深.NET程序员，全新视觉，深入解读ASP.NET 4.0核心开发技术

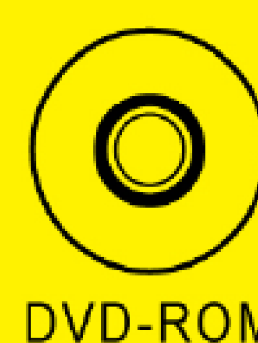
精通ASP.NET 4.0网络编程

——基础、框架与项目实战

（36小时多媒体教学视频）

孙继磊　等编著

◎ 技术与真实案例结合，源码管理→三层架构→单元测试，步步为营
◎ 大型项目与软件工程相结合，需求→设计→实现→测试，有条不紊
◎ 深入剖析LINQ技术与Entity Framework、ASP.NET AJAX、jQuery等框架
◎ 注重实战，详细介绍3个源自一线的项目开发实战案例的实现过程
◎ 对每章的重点内容录制了多媒体语音教学视频，高效、直观



清华大学出版社

精通 ASP.NET 4.0 网络编程

——基础、框架与项目实战

孙继磊　等编著

清华大学出版社

北　京

内 容 简 介

本书从实战出发，全面、系统地介绍了微软新发布的 ASP.NET 4.0 网络开发基础、相关开发框架及应用。书中提供了大量实例，并提供了 1 个通用模块和 3 个源自一线的项目开发案例供读者实战演练。本书附带 1 张光盘，内容为本书涉及的源代码和配套的教学视频，另外还赠送了 C#、ASP.NET 入门教学视频等其他学习资料。

本书共分 3 篇。第 1 篇介绍了模板页、主题、Web 服务、用户控件、自定义控件、ADO.NET 数据库访问技术，ASP.NET 数据控件、源码管理、三层结构、单元测试及搜索引擎优化等 Web 开发的关键技术；第 2 篇介绍了 Visual Studio 2010 新特性、LINQ 与实体框架 Entity Framework、ASP.NET AJAX 框架、优秀的 JavaScrip 框架 jQuery 等内容；第 3 篇介绍了 1 个通用权限管理系统的开发，另外，重点介绍了县长公开电话受理系统、社保卡结算系统和新农合管理系统 3 个实际项目的开发过程，这 3 个项目都是作者开发的拥有知识产权的项目，对提高读者的项目开发实战水平有很大帮助。

本书内容丰富，重点突出，适合有 C#语言基础的 ASP.NET 网络开发人员阅读，尤其适合想提高实际项目开发水平的人员阅读。另外，本书实用性强，很适合相关培训学校的学员作为教材使用。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目（CIP）数据

精通 ASP.NET 4.0 网络编程：基础、框架与项目实战 / 孙继磊等编著. —北京：清华大学出版社，2011.1
ISBN 978-7-302-24122-5

Ⅰ. ①精…　Ⅱ. ①孙…　Ⅲ. ①主页制作－程序设计　Ⅳ. ①TP393.092

中国版本图书馆 CIP 数据核字（2010）第 232254 号

责任编辑： 夏兆彦
责任校对： 徐俊伟
责任印制：
出版发行： 清华大学出版社　　**地　　址：** 北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62795954，jsjjc@tup.tsinghua.edu.cn
质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销： 全国新华书店
开　　本： 185×260　　**印　　张：** 36.5　　**字　　数：** 908 千字
（附光盘 1 张）
版　　次： 2011 年 1 月第 1 版　　**印　　次：** 2011 年 1 月第 1 次印刷
印　　数： 1～ 000
定　　价： 　　元

产品编号：040275-01

前　言

为什么要写这本书？

目前市场上 ASP.NET 类的图书很多，也有少数可以称得上经典和精品的图书，但是能够将技术与实际项目开发很好地结合起来讲解的书却是凤毛麟角。这主要是由于真实项目大都涉及软件公司或者用户的知识产权和商业机密，不能公开出版。因此，我们在市场上见到的各种 ASP.NET 开发类图书中的例子，尤其是最后的综合案例，都是类似于教学案例性质的演示程序而非实际应用案例。例如，我们经常看到的论坛系统、网上书店、网上购物、医院管理系统等。这种案例规模小，功能不全面，界面不细腻，软件可靠性不高。从各个技术角度来说，这种以教学和演示为目的的案例与真实项目开发都有很大差距。

当然，各种类型的 ASP.NET 图书都有自己的长处和适合的读者定位，如前面所说的简化的以教学为目的的案例，其优势在于门槛低，涉及技术少，程序结构简单，容易理解，适合于当学生教材或者给没有软件开发基础的读者阅读。而对于已经掌握了基本的 C#语言和 ASP.NET 基本语法的读者来说，他们更希望有一本能将具体技术和实际项目开发很好地结合起来，指导他们提高项目实战开发水平的书，这就需要以真实项目案例为背景指导读者学习。

为了帮助缺乏项目经验的读者深入理解真实的软件项目开发，笔者挑选了几个曾经做过的实际项目，从需求、设计、实现、测试几个过程进行讲解，帮助读者理解项目开发。为了让读者能比较好地理解项目开发，本书前半部分先重点介绍了相关项目中用到的 ASP.NET 开发技术和相关框架，最后提供了 1 个通用模块和 3 个源自一线的项目开发案例供读者实战演练。这 3 个案例都是作者近两年设计开发的拥有知识产权的真实项目，对提高读者的项目开发实战水平有很大帮助。

本书的写作和出版受到滨州学院科研基金的赞助，项目编号为 BZXYG0905。

本书有何特色？

1．紧跟行业发展，关注最新技术

本书对 Visual Studio 2010 /ASP.NET 4.0/ C# 4.0 /ADO.NET 4.0 中出现的新技术进行了讲解，如集成开发环境新功能、C# 4.0 新特性、LINQ、Entity Framework、AJAX 等。

2．技术全面，讲解深入、透彻

本书比较全面、系统地介绍了 ASP.NET 网络编程所涉及的关键技术，并对所涉及的

第三方框架做了重点深入、透彻的讲解。这些内容并不只局限于 ASP.NET 本身，还涉及其他客户端脚本技术、软件设计思想、软件开发规范等知识。

3．内容有所取舍，做到重点突出

本书不讲解 C#语言基础和太多的 ASP.NET 语法基础，而是关注于控件和页面的高级应用及实现原理，尤其对各种开发技术在实际项目开发中的应用做了重点介绍。

4．精选真实项目案例，提供完整的源代码，超级实用

本书精选了 1 个阶段性的项目案例（网上书店）、1 个通用模块系统（通用权限管理系统）和 3 个拥有自主知识产权的真实项目案例（县长公开电话受理系统、社保卡结算系统、新农合管理系统）进行讲解，并提供了完整的源代码，内容非常实用。通过这些案例，读者可以深入理解项目开发的过程，提升项目开发水平。

5．配超值 DVD 光盘

本书配带 1 张非常超值的 DVD 光盘，内容如下：

- ❑ 本书配套多媒体教学视频；
- ❑ 本书所涉及的源代码；
- ❑ C#入门教学视频（免费赠送）；
- ❑ ASP.NET 入门教学视频（免费赠送）；
- ❑ 其他学习资料（免费赠送）。

说明： 配书光盘中提供了县长公开电话受理系统、社保卡结算系统两个项目的全部源码。新农合项目只提供了部分代码。因为该系统包含两个相对独立的子系统，其中一个子系统是使用 WinForm 开发的，该部分内容书中没有介绍，所以没有提供该部分的源代码。

本书内容及知识体系

第1篇　ASP.NET网络开发关键技术（第1~6章）

本篇介绍了 ASP.NET 网络编程必须要掌握的一些关键技术。主要内容包括 ASP.NET 网络开发基础知识，如 ASP.NET 事件模型、页面生命周期、模板页、主题、Web 服务、用户控件、自定义控件等；ADO.NET 数据库访问技术，如连接数据库、修改数据、查询数据、储存过程等；ASP.NET 数据控件，如 GridView 控件、DataList 控件、数据源控件等；1 个阶段性项目案例网上书店的实现；规范的软件开发，介绍管理源码、三层结构和单元测试等；搜索引擎优化，介绍 URL 重写优化、正则表达式与 URL 重写、页面内容优化等。

第2篇　开发工具与第三方框架（第7~10章）

本篇介绍了 ASP.NET 网络开发所涉及的开发工具和第三方框架的使用。主要内容包括 Visual Studio 2010/C# 4.0/ASP.NET 4.0 的新功能和新特性，如集成开发环境的改进、C#

对动态数据类型的支持、ASP.NET 4.0 中的配置文件转换等；LINQ 和实体框架 Entity Framework 的使用，这是微软公司推出的最新的数据访问框架，提出了集成于语言中的与具体数据源相分离的数据访问和查询技术，大大提高了开发人员的效率；AJAX 框架原理、ASP.NET 自带的 ASP.NET AJAX 核心组件的使用、微软公司提供的 AJAX Control Toolkit 中的几种典型控件的使用；通过 JavaScript 框架 jQuery 实现丰富的动态页面效果，以及用 jQuery+ASP.NET Web Service 构建 AJAX 应用等。

第3篇　项目实战（第11~14章）

本篇综合利用前面所介绍的技术和思想，讲解了 4 个真实项目案例的设计与实现。主要内容包括通用权限管理系统，可以不经修改即可应用于各个 ASP.NET 项目，实现基本的基于角色的权限管理；县长公开电话受理系统，能够对县长公开电话工作进行全面的业务处理、数据查询、统计、报表等；社保卡结算系统，用于实现各个定点医疗机构的社保卡结算和对账功能，也包括各种数据查询、统计和报表等；新农合管理系统，能够对新型农村合作医疗业务进行日常管理，如农民档案管理、参合退合管理、缴费管理、报销结算等。

适合阅读本书的读者

本书假定读者已经具备了一定的编程基础，掌握了 C#语言、SQL 语句和 SQL Server 数据库的使用，所以书中没有涉及太多的基本语法、基本控件和基本开发环境操作等内容的讲解，而是把重点放在了关键技术、框架和项目实战上。如果您还不具备相关的基础，请首先阅读相关书籍，打好基础，才能比较流畅地阅读本书。本书适合的读者如下：

- 具备基本的 ASP.NET 知识，想进一步学习和提高的人员；
- 开发过 C/S 结构程序，想学习 B/S 开发的人员；
- 使用 ASP.NET 开发 Web 应用的程序员；
- 想提高 Web 项目开发水平的程序员；
- 大中专院校和培训班的学生。

本书作者及编委会成员

本书由孙继磊主笔编写。其他参与编写的人员有班志杰、陈旭、陈永俊、陈争光、戴建华、方文票、冯玉荣、高姗姗、巩宁来、谷世江、胡其吐、黄飞龙、蒋晓捷、李德明、李显亮、李志勇、刘雁征、吕小波、马东、孟庆海、唐勇、王浩、王玲玉、王志娟、武娜、徐晓娟、闫树丰、杨朝宇、翟闯等。在此表示感谢！

本书编委会成员有欧振旭、陈杰、陈冠军、项宇峰、张帆、陈刚、程彩红、毛红娟、聂庆亮、王志娟、武文娟、颜盟盟、姚志娟、尹继平、张昆、张薛。

编著者

目　　录

第 1 篇　ASP.NET 网络开发关键技术

第 2 篇　开发工具与第三方框架

第 3 篇 项目实战

第 1 篇　ASP.NET 网络开发关键技术

第 1 章　ASP.NET 网络开发基础

本章将介绍 ASP.NET 的几个关键技术，包括服务器端控件事件模型、页面生命周期、母版、用户控件和自定义控件等。阅读本书的读者应该对 ASP.NET 有基本的了解，熟悉 Visual Studio 开发环境，能够使用 Visual Studio（或者 Visual Web Developer Express）创建 ASP.NET Web 应用程序并且运行，了解 ASP.NET 常用的基本控件，如 Button、TextBox、DropDownList 等。

1.1　ASP.NET 事件模型和页面生命周期

ASP.NET 对 Web 页面控件的事件进行了封装，从而允许开发人员用一种类似于处理本地 WinForm 窗体事件的方式来处理 Web 页面事件，大大简化了开发流程。任何事物都有两面性，ASP.NET 这种封装也不例外，在带来极大方便的同时，也使得 ASP.NET 页面从创建到销毁的过程更加复杂。

本节将对 ASP.NET 服务器端控件事件模型和 ASP.NET 页面生命周期进行介绍，掌握这两个知识点对于深入理解 ASP.NET 运行原理、进行 ASP.NET 高级开发有重要作用。

1.1.1　经典的 Web 事件处理方法

为了理解 ASP.NET 服务器控件事件模型及其优点，先要理解 Web 应用程序的请求——响应模型及事件处理方式。

Web 应用程序或者网站包含两个重要角色：服务器和浏览器（客户端）。用户通过浏览器向 Web 服务器发送 HTTP 请求，服务器响应此请求并且向浏览器返回 HTML 代码，HTML 代码到达浏览器后在浏览器中以可视化的形式显示出来，这个过程如图 1.1 所示。

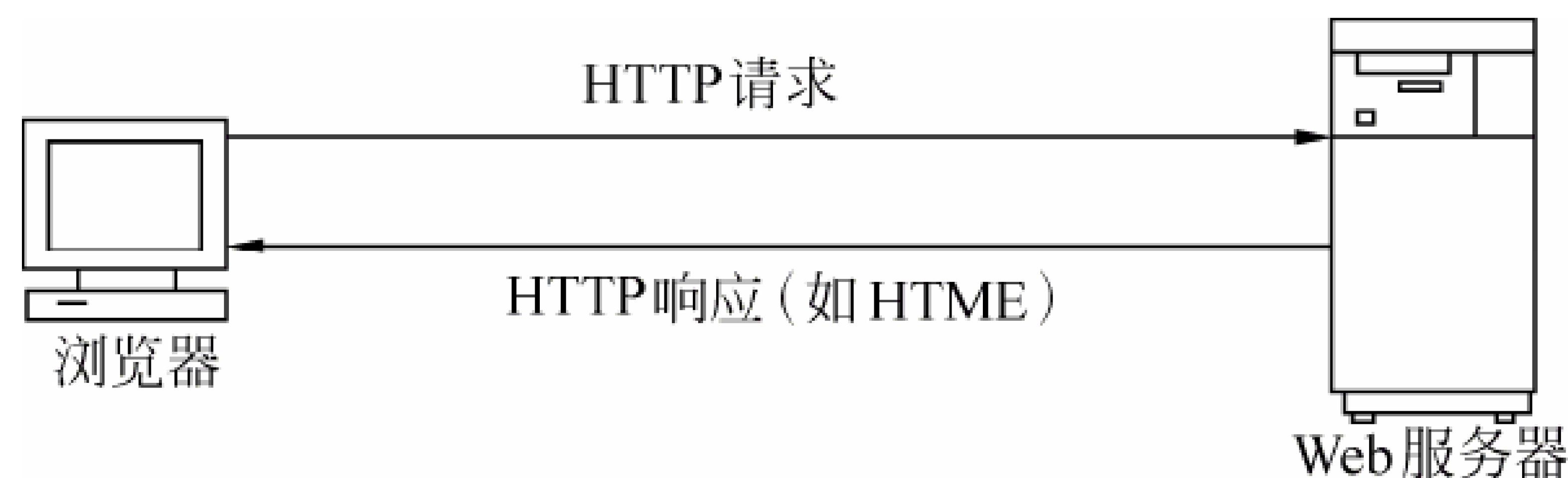


图 1.1　Web 请求——响应模型

用户在浏览器中进行的操作，如单击一个按钮等，也会转化为一个 HTTP 请求（通常是一个 POST）发送到服务器，然后 Web 服务器处理此请求并且返回响应内容。这个过程与前面所述的请求——响应过程完全相同。

对于 ASP.NET 开发人员来说，所开发的 ASP.NET 程序部署在 Web 服务器上，通过某种机制（通常是 IIS）响应和处理客户端请求。从上面的讨论可以看出，在 Web 应用程序中，开发人员需要针对各种不同的 HTTP 请求来编写代码，而与浏览器端的事件（如按钮的单击）无关。这种编程方式很不直观，客户端（浏览器）事件与服务器端事件的处理是完全分开的，下面通过一个例子具体说明此问题。

【例 1-1】 经典的 Web 事件处理。

本例演示在传统 Web 编程中如何处理客户端事件。在本例中，用户在浏览器端单击一个按钮，则服务器响应此按钮的单击事件并返回服务器当前时间。

完成这个例子的具体操作过程如下。

（1）在 Visual Studio 中创建一个 ASP.NET Web 应用程序，项目命名为 ClassicWebApp。项目创建后，会自动生成一个 Default.aspx 的 ASP.NET Web 窗体，由于本例采用基于纯粹的 HTTP 请求的方式处理页面事件，而不使用 ASP.NET Web 窗体，所以把 Default.aspx 从项目中删除。

提示： 本书假定读者已经熟悉了 Visual Studio 开发环境和基本操作，如创建项目、运行项目等，为了突出重点和节省篇幅，书中对于此类基本操作不给出具体操作过程和界面截图。如果读者对此不熟悉，可参考其他 ASP.NET 编程资料。

（2）在项目中添加一个“一般处理程序”。ASP.NET 的一般处理程序是一个实现了 IHttpHandler 接口的类，可以处理 HTTP 请求。

添加一般处理程序的具体操作过程是：选择菜单中的“项目”|“添加新项”，在弹出的“添加新项”对话框中选择“一般处理程序”，并重命名为 MyHandler，如图 1.2 所示。

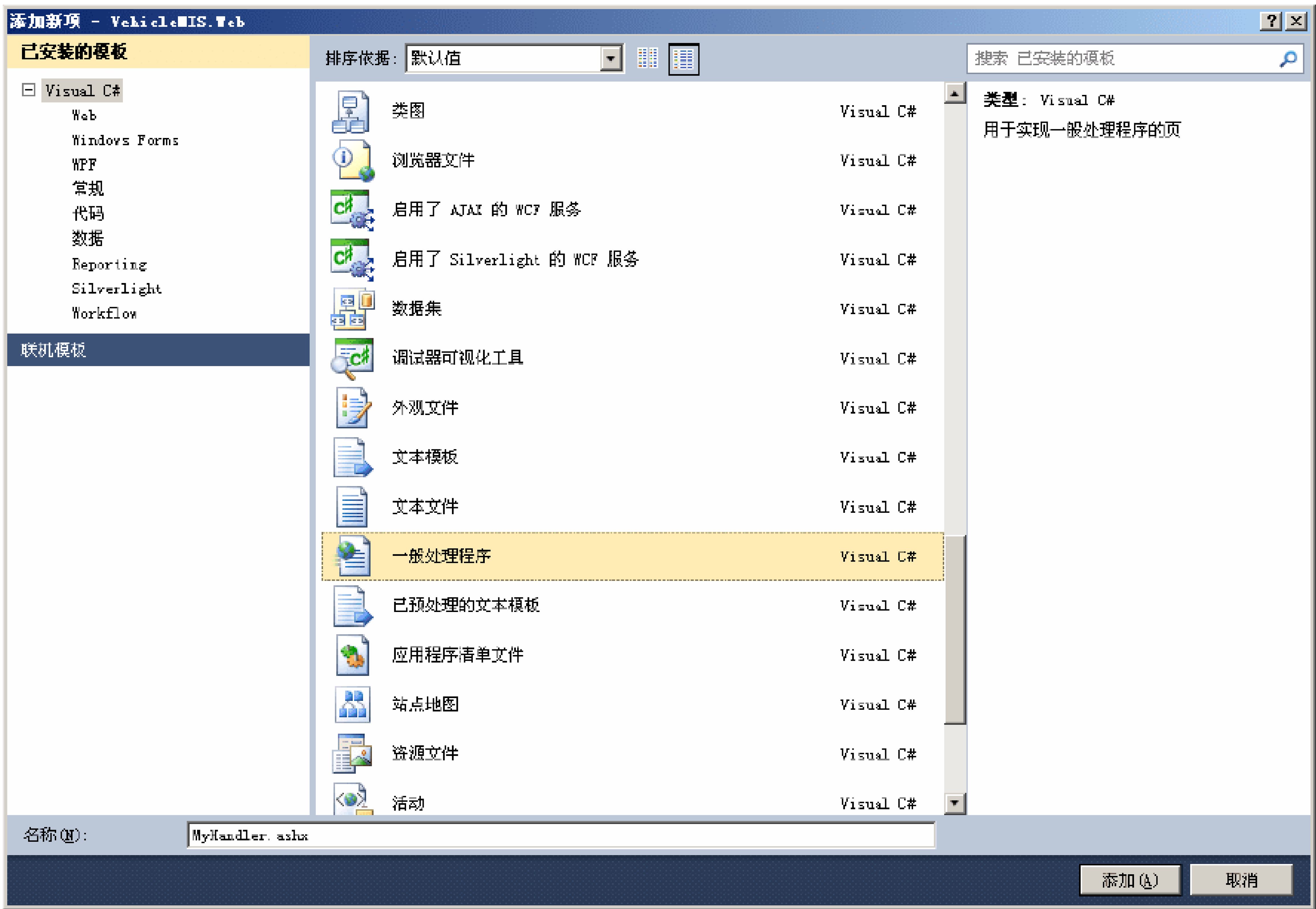


图 1.2　添加一般处理程序

（3）添加了一般处理程序 MyHandler 类后，可以看到其中有一个 ProcessRequest()方法，代码如下：

```
public void ProcessRequest(HttpContext context)
{
    context.Response.ContentType = "text/plain";
    context.Response.Write("Hello World");
}
```

这个方法就是用于处理浏览器发送过来的 HTTP 请求的方法。方法有一个 HttpContext 类型的参数，代表了当前 HTTP 请求上下文，可以通过这个属性向客户端返回特定的内容。本例向客户端返回服务器的当前日期和时间，如下代码所示。

```
public void ProcessRequest(HttpContext context)
{
    //设置 HTTP 响应类型为 HTML 文本
    context.Response.ContentType = "text/html";
    //构建一个 HTML 格式的包含当前服务器时间的字符串
    string s = string.Format("<html> <body> <h3> 服务器当前时间为：<span style
    = \"color:green;\" > {0:G} <span> </h3> </body> </html>", DateTime.Now);
    context.Response.Write(s);              //向浏览器返回服务器这个字符串
}
```

（4）在项目中添加一个 HTML 页面，命名为 MyPage.htm。

（5）在 MyPage.htm 中，添加一个表单 Form，并设置其 Action 属性为 MyHandler.ashx，从而使表单提交给 MyHandler.ashx，在表单中放一个提交按钮，以执行提交并获取服务器当前时间。MyPage.htm 页面的关键代码如下：

```
<body>
<form action="MyHandler.ashx">
<input type="submit" value="获取服务器时间" />
</form>
</body>
```

（6）右击 MyPage.htm 页面空白处，从弹出的快捷菜单中选择“在浏览器中查看”选项，则可以运行此页面，运行结果如图 1.3 所示。

（7）在 MyPage.htm 页面上单击“获取服务器时间”按钮，则浏览器会跳转到 MyHandler.ashx 页面（见提示），显示结果如图 1.4 所示。

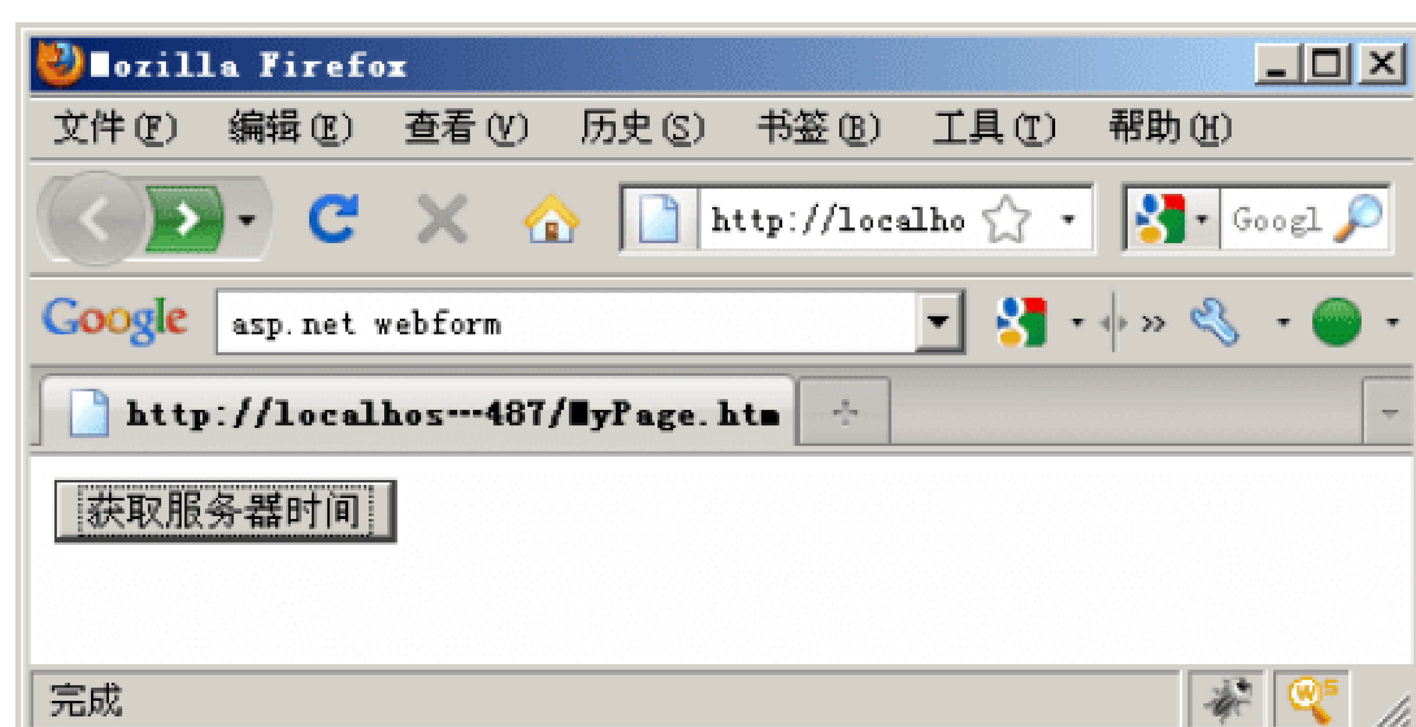


图 1.3　MyPage.htm 运行界面

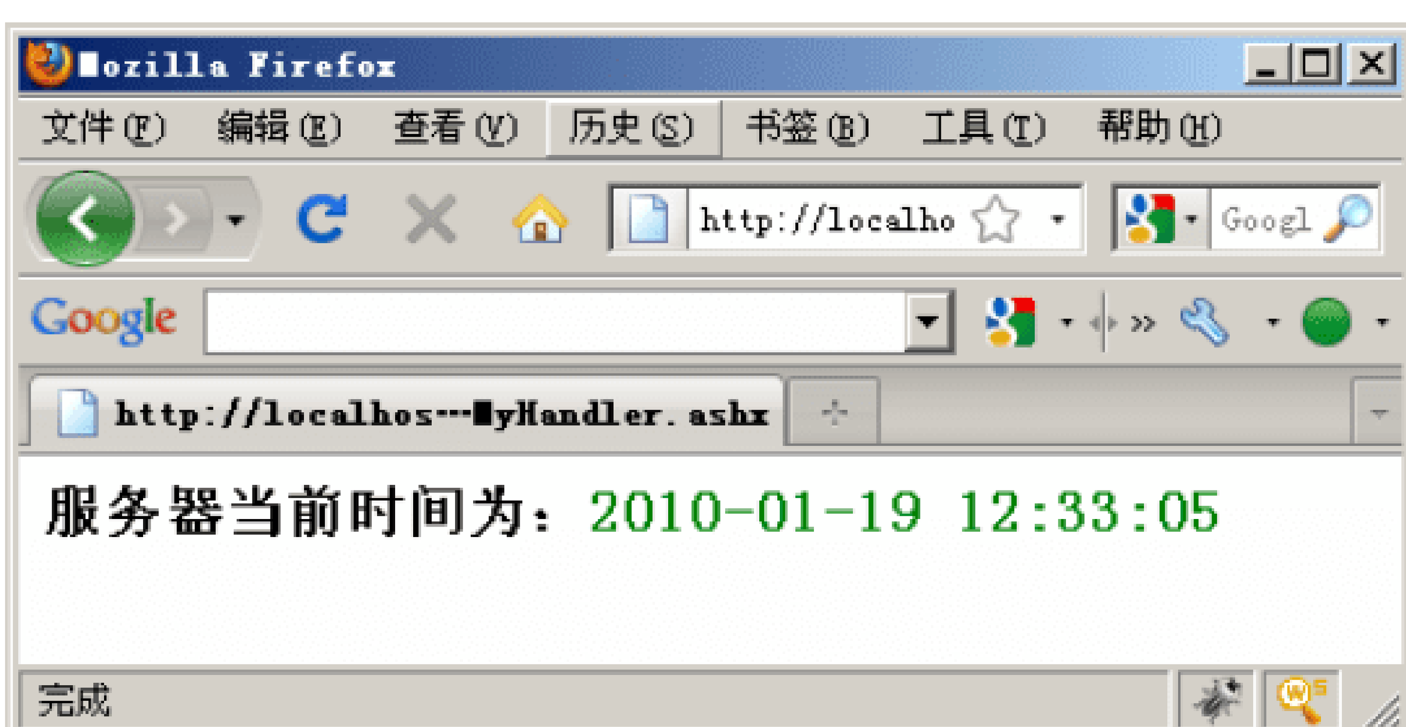


图 1.4　MyHandler.ashx 页面

提示：严格来说，MyHander.ashx 并不是一个普通意义上的页面，但是在浏览器看来，这是一个合法的 URL，可以访问并且能够得到用于显示的 HTML 代码，与其他页面没有本质区别。

在例 1-1 中，MyHanlder.ashx 处理 HTTP 请求时，总是返回服务器当前日期和时间，

而没有考虑 MyPage.htm 页面上的内容。在实际应用中，服务器端通常需要根据页面上的内容做出相应的处理。例如，对于用户登录程序来说，服务器后台处理程序需要获取用户在登录页面中输入的用户名和密码，进而判断是否合法用户。还有一些页面，其中包含多个按钮，而每个按钮的作用是不同的，服务器端需要区分用户在浏览器中单击了哪个按钮，从而进行不同的处理。

对于 ASP.NET Web 应用程序来说，服务器端代码可以通过 HttpContext.Request.Params 属性获得用户在浏览器页面中输入的内容，下面通过一个例子来演示具体实现。

【例 1-2】 处理浏览器端输入事件。

本例演示在 ASP.NET 中如何在服务器端获得浏览器页面上输入的内容，并根据用户单击不同的按钮而执行不同操作。

（1）创建一个 ASP.NET Web 应用程序，删除默认的 Default.aspx 页面。

（2）在项目中添加一个用户登录 HTML 页面，命名为 login.htm，关键代码如下：

```
<body>
<form action="LoginHandler.ashx">
用户名：<input type="text" name="username" /><br />
密码：<input type="password" name="password" /><br />
<!--注意：下面这两个按钮的 name 要相同，以便于在服务器端获取单击了哪个按钮-->
<input type="submit" name="button1" value="登录" />
<input type="submit" name="button1" value="注册" />
</form>
</body>
```

（3）在项目中添加一般处理程序，命名为 LoginHandler，为其 ProcessRequest()方法编写代码，根据不同情况处理 Web 请求。

```
public void ProcessRequest(HttpContext context)
{ //获取页面中输入的用户名，其中 Params 后面的键为页面上控件的名称
    string user = context.Request.Params["username"];
    string pass = context.Request.Params["password"];  //获取页面中输入的密码
    string button = context.Request.Params["button1"]; //获取所单击的按钮
    context.Response.ContentType = "text/plain";       //设置响应类型为纯文本
    //如果单击“注册”按钮引起的提交，则提示功能未实现
    if (button == "注册")
    {
        context.Response.Write("注册功能尚未实现...");
    }
    //如果单击“登录”按钮引起提交，则判断输入的用户名和密码是否正确
    else if (button == "登录")
    {
        if (user.ToLower() == "sunjilei" && pass == "123456")
            context.Response.Write("登录成功！");
        else
            context.Response.Write("用户名或密码不正确，登录失败！");
    }
    //如果不是单击“注册”或“登录”按钮，则认为是不可识别的命令
    else
    {
        context.Response.Write("LoginHandler 不能识别接收到的命令。"+button);
    }
}
```

（4）运行项目，运行结果如图 1.5 所示。

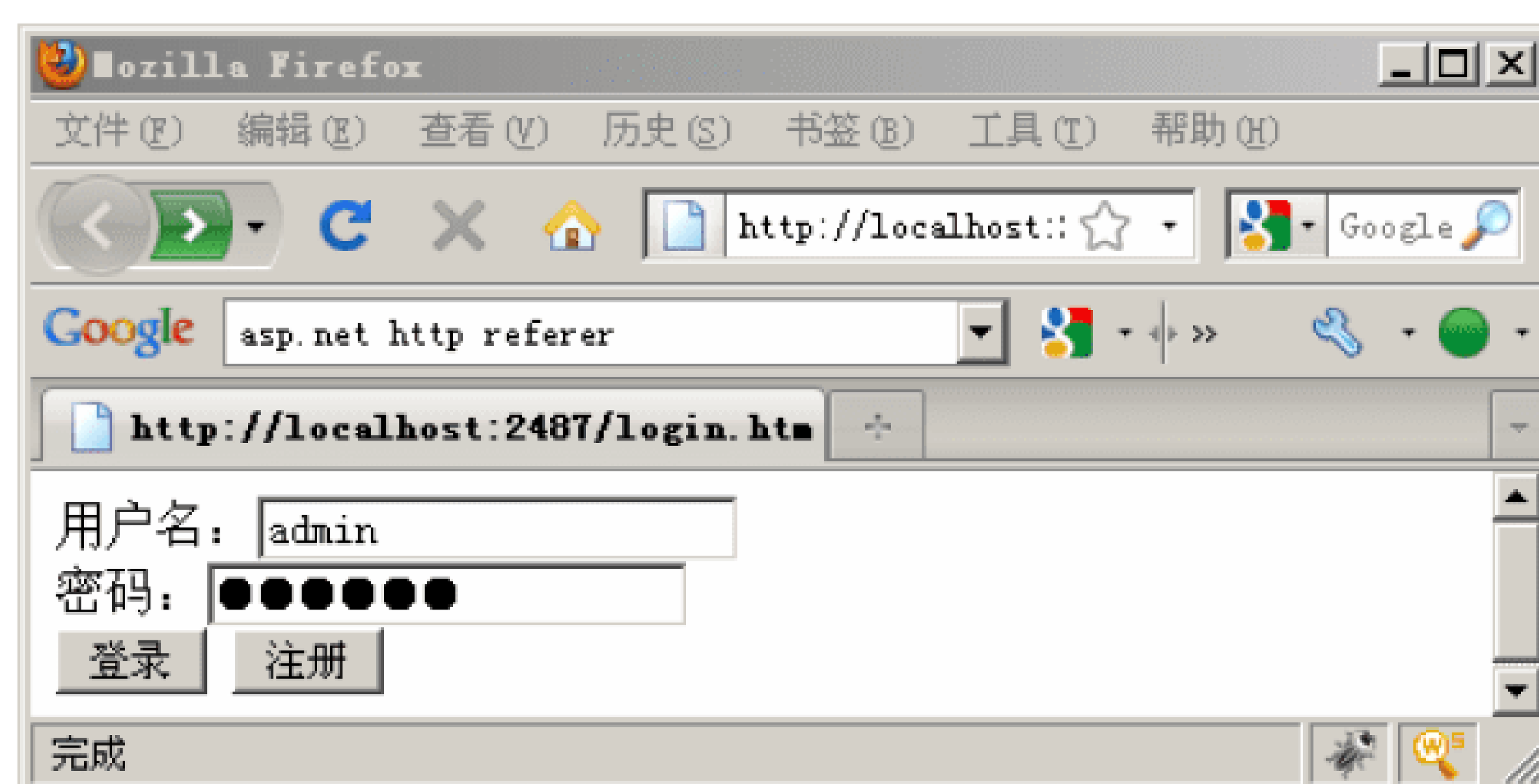


图 1.5　例 1-2 运行界面

在上述代码中，有两个地方需要说明。

（1）可以通过 context.Response.Params["username"]得到用户在浏览器页面中输入的登录用户名，其中 username 为 HTML 页面上用户名文本控件的名称。可以通过同样的方法得到页面上其他控件的值。

（2）由于此页面有两个提交按钮，一个“登录”，一个“注册”，两者功能不同。在同一个处理程序 LoginHandler 中，必须区分这两个按钮。这里使用了一个小技巧，即把两个按钮的名称（name 属性）设置为相同（都是 button1），而按钮的文本（value 属性）不相同，然后在服务器端就可以通过获取 button1 控件的值来判断单击了哪个按钮。

1.1.2　ASP.NET 服务器控件事件模型

从 1.1.1 节的两个例子可以看出，在传统的基于请求——响应的 Web 事件处理模型中，一个页面所有控件的事件通常都由一个程序来处理，事件处理程序与事件之间没有直接联系。另一方面，要想获得页面上输入的内容也不方便，需要采用 Request.Params["控件名"]这种形式。由于控件名是一个字符串，不能利用智能提示，不能进行编译检查，所以很容易写错。

ASP.NET 的服务器控件事件模型封装 Web 请求——响应的底层细节，从而很好地解决了上述问题，使控件事件与其处理程序紧密结合，使得控件的访问更安全高效。

ASP.NET 服务器端控件事件模型与传统的 WinForm 事件类似，页面上有许多控件，一个控件有许多事件，对于每个控件的每个事件都可以单独编写一个事件处理程序，当控件发生某个事件时，就会执行对应的事件处理程序。

下面通过一个例子演示 ASP.NET 事件模型的优势。

【例 1-3】 登录页面。

本例与例 1-2 功能相同，所不同的是本例采用 ASP.NET Web 窗体+服务器端控件来实现。对比同一功能的两种不同实现，更能体现二者差别。

（1）创建一个 ASP.NET Web 应用程序，命名为 EventModelSample。

（2）在自动生成的 Default.aspx 上放置两个 TextBox 控件和两个 Button 控件，页面布局与例 1-2 相同。页面代码如下：

```
<body>
   <form id="form1" runat="server">
   <div>
   用户名：<asp:TextBox ID="username" runat="server"></asp:TextBox><br />
   密码：<asp:TextBox ID="password" TextMode="Password" runat="server">
   </asp:TextBox><br />
      <asp:Button ID="login" runat="server" Text="登录" onclick="login
      Click" />
      <asp:Button ID="register"
         runat="server" Text="注册" onclick="register_Click" />
   </div>
   </form>
</body>
```

（3）为“登录”按钮和“注册”按钮编写事件处理程序，代码如下：

```
//登录按钮事件处理程序
protected void login_Click(object sender, EventArgs e)
{
   //获取用户输入的用户名和密码
   string user = username.Text;
   string pass = password.Text;
   if (user.ToLower() == "sunjilei" && pass == "123456")
      Response.Write("登录成功！");
   else
      Response.Write("用户名或密码不正确，登录失败！");
}
//注册按钮事件处理程序
protected void register Click(object sender, EventArgs e)
{
   Response.Write("注册功能尚未实现...");
}
```

对比例 1-3 和例 1-2 可以看出，在 ASP.NET Web 窗体中，每个控件的事件都是单独的方法，各个事件处理程序相互独立，事件与其处理程序有明确的对应关系，从而使代码更加清晰简洁，提高了可维护性。

1.1.3　ASP.NET 页面生命周期

一个 ASP.NET 页面，从创建到销毁的过程，称为页面的生命周期。这是一个短暂但却复杂的过程。当 ASP.NET 接收到对于某一页面的请求后，此页面就被创建，从而开始了页面的生命周期，经过一系列操作后，页面产生了对应的 HTML 代码并且返回给浏览器，页面就被销毁，页面生命周期也就结束。

一个 ASP.NET 页面是一个 System.Web.UI.Page 类的派生类，在页面从创建到销毁的生命周期中，Page 类会先后触发多个事件。表 1.1 按照发生的先后顺序列出了 Page 类的主要事件。

表 1.1　Page类生命周期主要事件

事 件 名 称	说　明
PreInit	页面初始化以前触发，通常可以在此事件中检查 IsPostBack 属性，动态创建控件或者动态设计主题
Init	当页面中所有控件初始化完成时触发，可以在此事件中访问控件初始化属性

续表

事 件 名 称	说　　明
InitComplete	初始化完成后触发，到此事件发生时页面上所有控件和页面本身的初始化都已完成
PreLoad	在页面加载前触发，在此事件中会加载页面和控件的视图状态（ViewState），并处理回发数据
Load	页面加载事件，可以此事件中设置控件属性
LoadComplete	控件及页面加载完成后触发此事件
PreRender	在页面即将呈现时触发，可以此事件中对页面或者控件外观进行最后的修改
Unload	页面处理完时触发，通常在此事件中进行清理工作

在 ASP.NET 页面生命周期中，要完成一些工作，包括加载主题、加载视图状态、呈现控件等。从 ASP.NET 页面要完成的工作角度看，页面生命周期如图 1.6 所示。

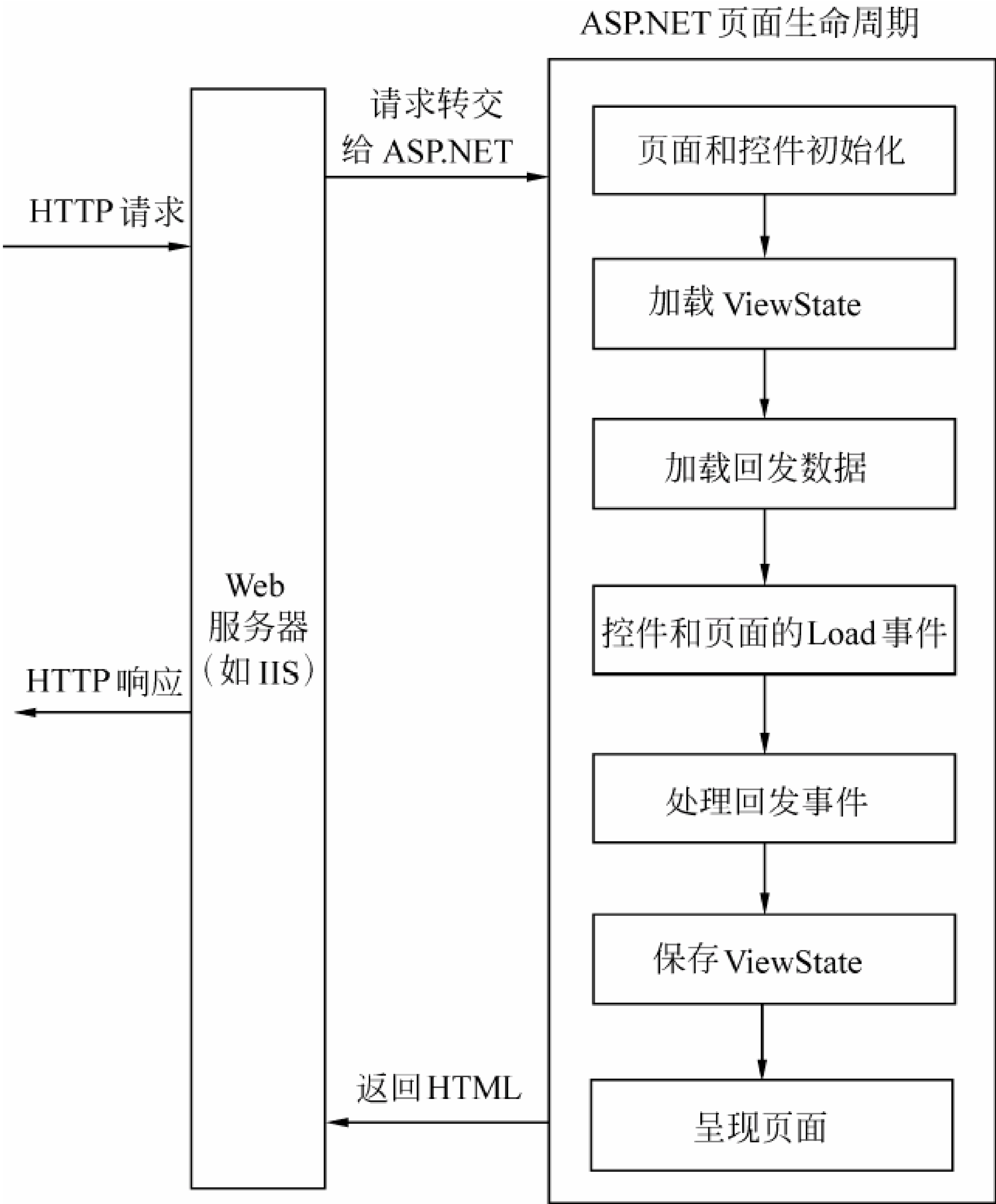


图 1.6　ASP.NET 页面生命周期的各阶段工作

下面通过一个例子来更加直观地说明页面及控件事件发生的顺序。

【例 1-4】 页面生命周期。

本例创建一个简单的页面，其中包含几个控件，当页面的主要事件发生时，在页面上输出提示信息，从而演示页面生命周期。

（1）创建一个 ASP.NET Web 应用程序，命名为 PageLifeCycle。

（2）在项目中添加一个页面，并参照以下代码修改页面。

```
<body>
    <form id="form1" runat="server">
    <div>
```

```
        输入你的名字：<asp:TextBox ID="yourName" runat="server"
            ontextchanged="yourName_TextChanged"></asp:TextBox>
        <asp:Button ID="ok" runat="server" Text="确定" onclick="ok_Click" />
        <br />
        <asp:Label ID="hello" runat="server" Text="Label"></asp:Label>
    </div>
    </form>
</body>
```

（3）为页面上按钮的 Click 事件和文本框的 TextChanged 事件添加处理程序，代码如下：

```
protected void ok_Click(object sender, EventArgs e)
{
    Response.Write("按钮 Click 事件<br/>");
    hello.Text = yourName.Text+"，你好。";
}
protected void yourName_TextChanged(object sender, EventArgs e)
{
    Response.Write("文本框 TextChanged 事件<br/>");
}
```

（4）重载 Page 类与页面生命周期相关的方法，以便在页面事件发生时输出提示信息，代码如下：

```
protected void Page_Load(object sender, EventArgs e)   //PageLoad 事件
{
        Response.Write("页面加载事件（Page_Load）<br/>");
}
protected override void OnLoad(EventArgs e)            //OnLoad()方法
{
    Response.Write("页面加载事件（OnLoad）<br/>");
    base.OnLoad(e);
}
protected override void AddedControl(Control control, int index)
                                                       //控件被添加时调用
{
    Response.Write(string.Format("控件被添加：{0}<br/>",control.GetType().
    Name));
    base.AddedControl(control, index);
}
protected override void LoadControlState(object savedState)
                                                           //加载控件状态
{
    Response.Write("加载控件状态(LoadControlState)<br/>");
    base.LoadControlState(savedState);
}
protected override void LoadViewState(object savedState)   //加载视图状态
{
    Response.Write("加载视图状态(LoadViewState)<br/>");
    base.LoadViewState(savedState);
}
protected override void OnInit(EventArgs e)                //页面初始化
{
    Response.Write("页面初始化事件（OnInit）<br/>");
    base.OnInit(e);
}
```

```
protected override void OnLoadComplete(EventArgs e)         //页面加载完成
{
    Response.Write("页面加载完成（OnLoadComplete）<br/>");
    base.OnLoadComplete(e);
}
protected override void OnPreLoad(EventArgs e)              //页面加载前
{
    Response.Write("页面即将加载（OnPreLoad）<br/>");
    base.OnPreLoad(e);
}
protected override void OnPreRender(EventArgs e)            //页面呈现前
{
    Response.Write("页面即将呈现（OnPreRender）<br/>");
    base.OnPreRender(e);
}
protected override void OnUnload(EventArgs e)               //页面卸载
{
    //在 Unload 事件中，Response.Write 已经被关闭，不可使用，下面代码不可执行
    //Response.Write("页面卸载（OnUnload）<br/>");
    base.OnUnload(e);
}
```

（5）运行此项目，运行结果如图 1.7 所示。

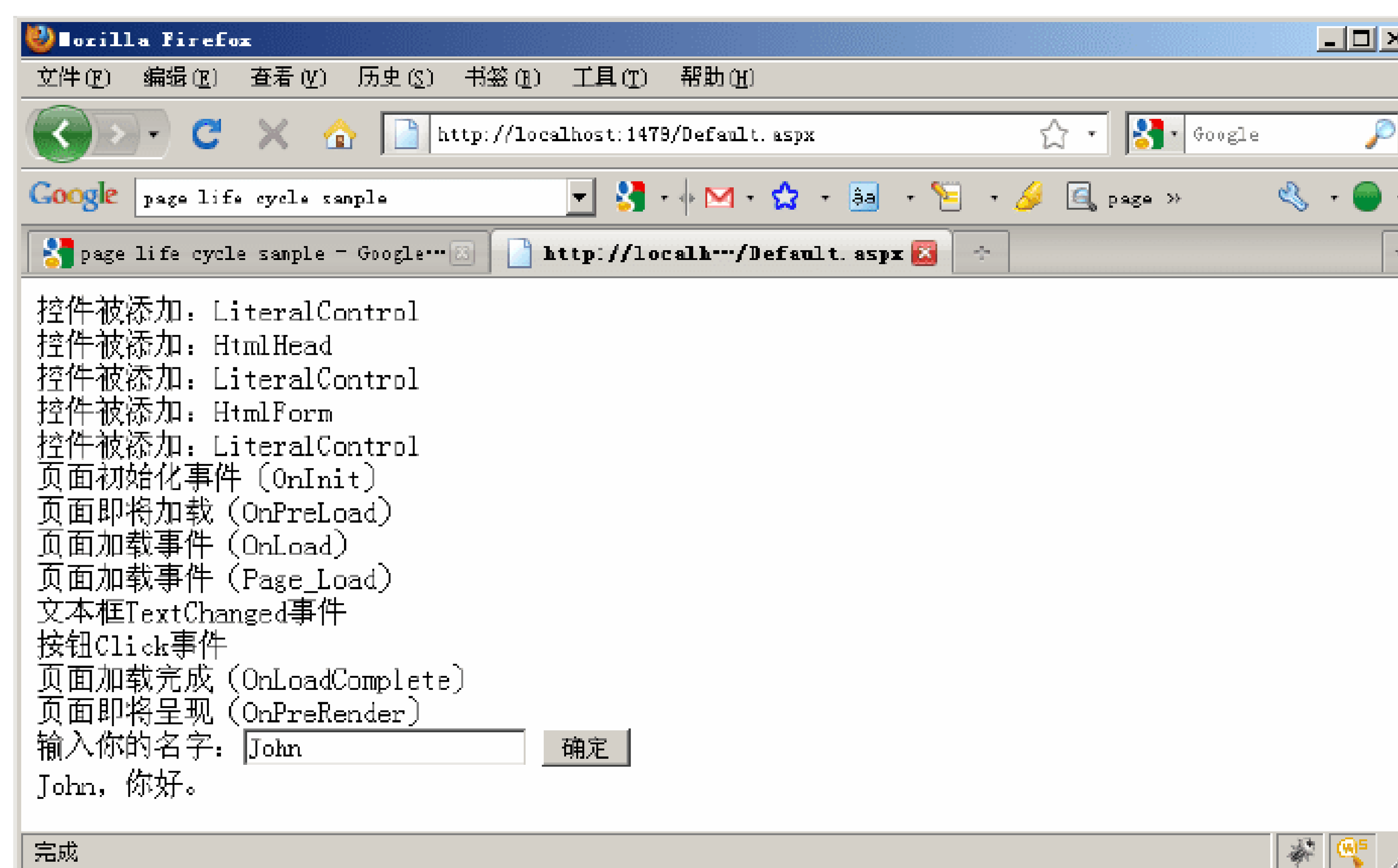


图 1.7　页面生命周期示例运行界面

1.2　母　版　页

不管是在做网站还是开发 Web 应用程序，为了使作品在外观上标准、规范，通常一个项目中的所有页面风格和结构是一致的，而且各个页面通常有一部分相同的地方，如公司 LOGO、页面顶部标题栏、页脚说明信息等。利用 ASP.NET 的母版技术，可以快速开发出风格一致且易于维护的页面。

1.2.1　母版页的概念和作用

如前所述，Web 应用程序通常要求许多页面具有外观上的一致性，例如具有相同的标

题栏、LOGO、页脚等。为了达到这种效果，最简单的做法是把相同的代码复制粘贴到每一个页面中。但是这显然不符合软件设计思想，相同的代码重复出现了多次，这会给以后的修改造成很大困难。

根据面向对象思想，如果许多事物（类）具有相同的功能，那么可以从这些事件（类）抽象一个基类，把公共的代码放到基类中，再从基类派生其他类，这样就实现了代码复用。而且如果要修改公共的功能，只需要修改基类一个类就可以。

把这种抽象和继承的思想应用于 ASP.NET 页面的设计，这就是母版页。母版页就相当于基类，其中包含着公共的元素，从母版页可以创建其他页面，相当于从基类派生，派生类自然就具有了基类的所有功能。

与基类在类层次结构中所起的作用类似，母版页在页面设计中就相当于一个模板或框架，在此基础上，可以创建其他页面。从母版页创建的页面称为“内容页”，内容页自动包含了母版页中所有的元素，而且还可以添加自己的元素以扩充母版的内容。

在类的继承中，基类中包含虚方法，派生类可以重写此方法。与此类似，母版页中包含称为“内容占位符”的控件 ContentPlaceHolder，内容页中可以在占位符控件中按照自己的要求放置各种控件。

母版页的扩展名为.master，一个母版页中可以包含一到多个占位符。当浏览器请求一个内容页时，ASP.NET 会把母版页与内容页合并，再把合并后的页面发给浏览器。这个过程如图 1.8 所示。

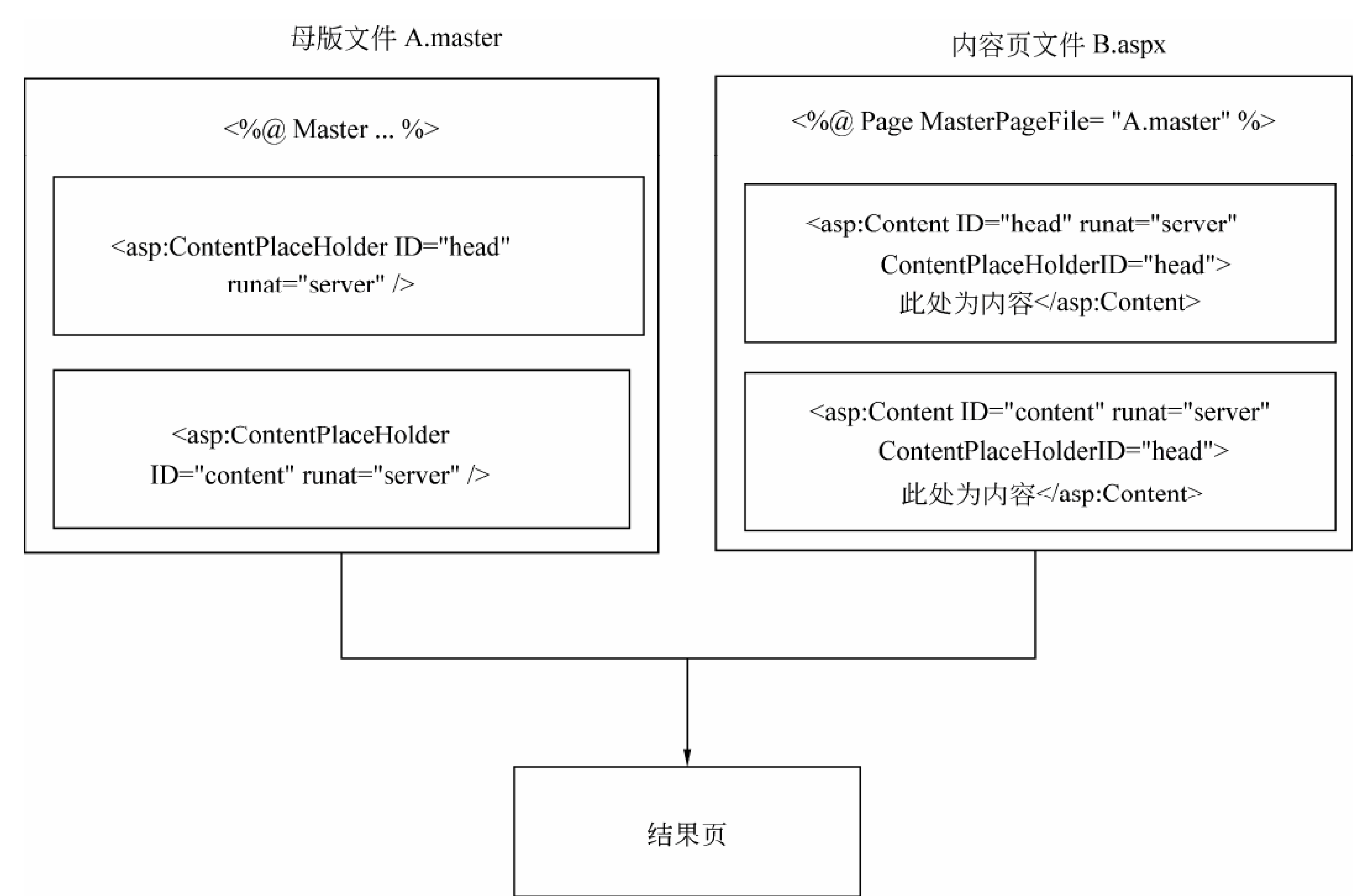


图 1.8　母版页与内容页

注意： 母版页必须与内容页合并后作为一个单独页面显示在浏览器中。母版页本身不能直接在浏览器中显示。

1.2.2　创建和使用母版页

创建母版页和使用母版页创建内容页的方法与创建普通的 ASP.NET 页面基本相同，下面通过一个例子来具体说明。

【例 1-5】 创建母版页和内容页。

本例模仿滨州学院网站创建了一个学校网站的母版，并从此母版创建了一个内容页。

（1）创建一个 ASP.NET Web 应用程序，重命名为 SchoolWebSite。

（2）从菜单中选择“项目”|“添加新项”命令，在弹出的“添加新项”对话框中选择“母版页”，添加一个母版页。

（3）模拟滨州学院网站首页，在项目中添加必要的图片，在母版页中添加一些页面顶部的图片和页面底部的文字，代码如下：

```
<body>
<form id="form1" runat="server">
<div>
<!--下面的 div 是母版中的内容，页顶部的图片和链接-->
<div>
<img src="images/bzxy.png" alt="滨州学院" /> <br />     <%--学校 LOGO--%>
<%--以下各个 a 标记是到学校各个部门的链接--%>
<a href="http://www.bzu.edu.cn/xuexiaogaikuang/20070821/59.html" class=
"nav">学校概况</a>|
<a  href="http://www.bzu.edu.cn/jiaoxuekeyan/20070830/222.html"  class=
"nav">教学科研</a>|
<a href="http://rsc.bzu.edu.cn/" class="nav">师资队伍</a>|
<a href="http://zs.bzu.edu.cn/" target="_blank" class="nav">招生就业</a>|
<a  href="http://www.bzu.edu.cn/guojijiaoliu/20070830/233.html"  class=
"nav">国际交流</a>|
<a  href="http://www.bzu.edu.cn/yuanxishezhi/20070913/349.html"  class=
"nav">院系设置</a>|
<a href="http://www.bzu.edu.cn/dangzhengjigou/20070830/237.html" class=
"nav">党政机构</a>|
<a  href="http://www.bzu.edu.cn/shetuanjigou/20070830/250.html"  class=
"nav">群团机构</a>|
<a href="http://hqgl.bzu.edu.cn/" target="_blank" class="nav">后勤服务</a>
</div>
<asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
<!--这里是占位符，内容页可以修改这里的内容。-->
</asp:ContentPlaceHolder>
<!--下面的 div 是母版中的内容，页脚，显示了联系方式、版权信息、网站备案-->
<div><table  align="center"  border="0"  cellpadding="0"  cellspacing="0"
width="931">
<tr><td class="end" background="/foot.jpg" height="80">
<div align="center">
<p>版权所有 &#169;  滨州学院 地址：山东省滨州市黄河五路 391 号 邮编：256600 电话：
0543-3190016 <br>
网站管理：滨州学院网络中心
<a href="http://www.miibeian.gov.cn/" target="_blank">鲁 ICP 备 07012503 号
</a>
<a href="http://www.bzga.gov.cn/" target="_blank">滨公备 0701173 号</a>
<a href="http://www.cyberpolice.cn/" target="_blank">网络报警</a></p>
</div></td> </tr>
```

```
</table> </div>
</div>
</form>
</body>
```

（4）从菜单中选择“项目”|“添加新项”命令，在弹出的“添加新项”对话框中选择“使用母版页的 Web 窗体”，如图 1.9 所示。

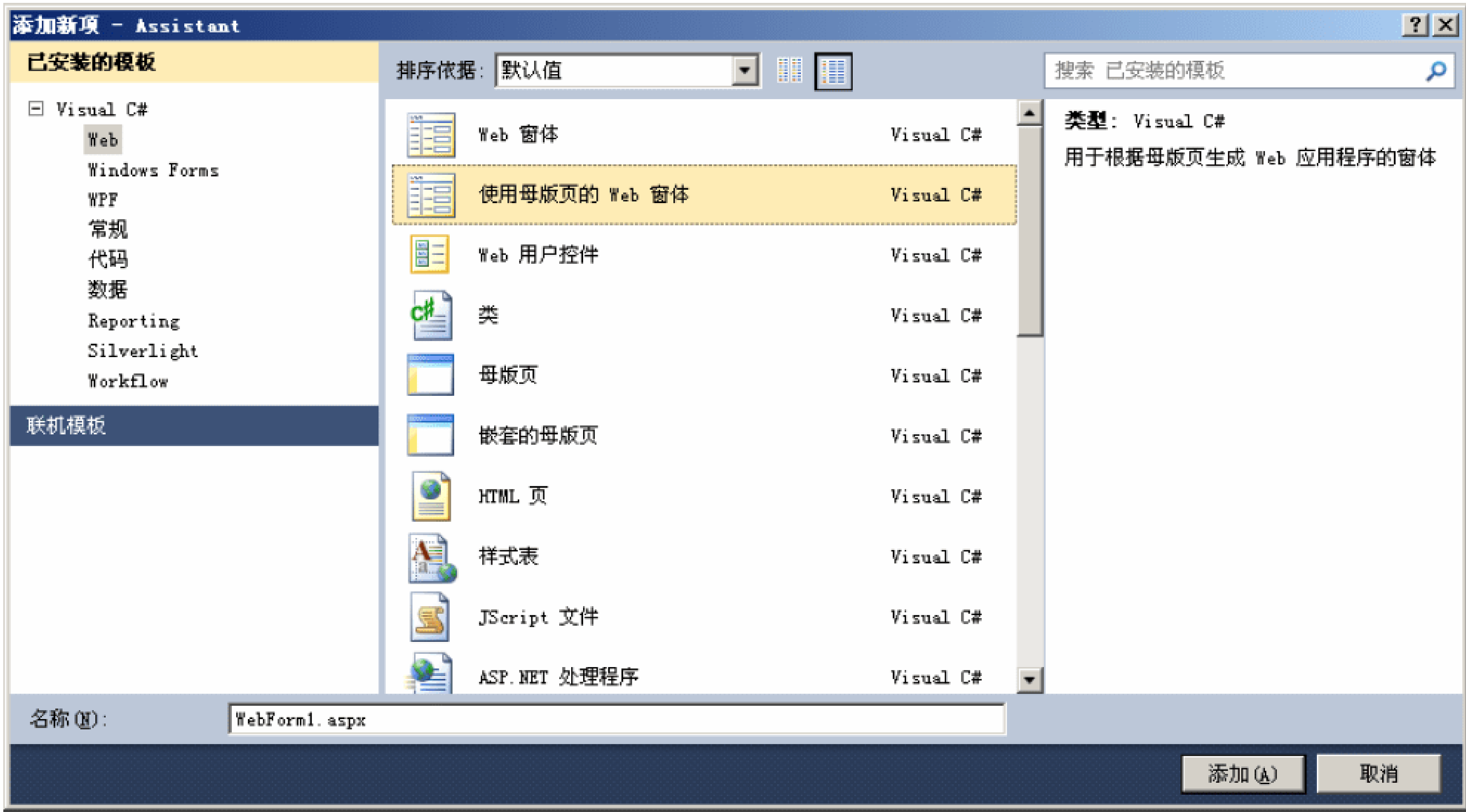


图 1.9　添加 Web 内容窗体

（5）在图 1.9 所示的对话框中选择了“使用母版页的 Web 窗体”并单击“添加”按钮以后，会出现如图 1.10 所示的对话框，要求选择一个母版页。选择上一步骤中所创建的母版页，单击“确定”按钮。

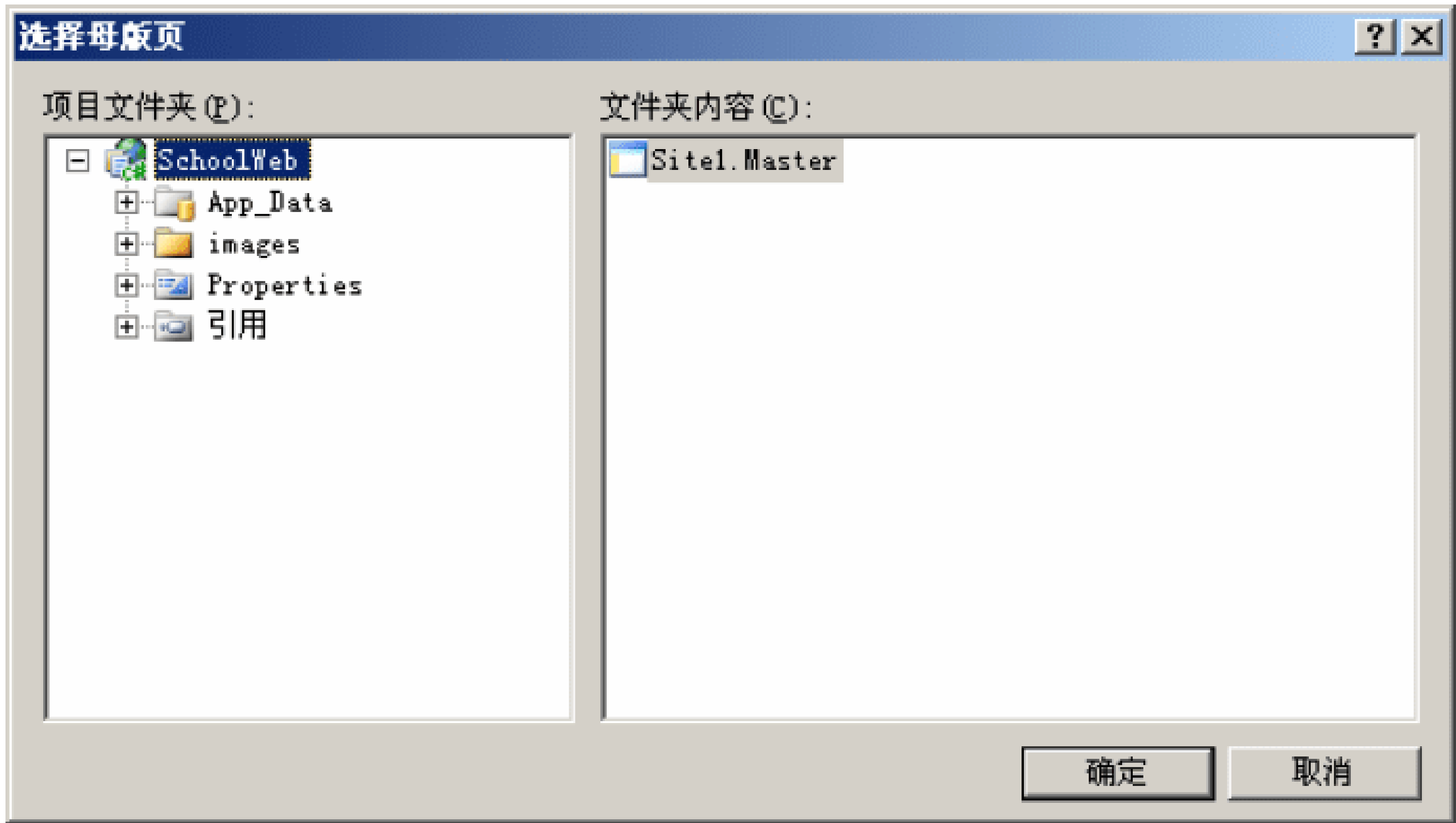


图 1.10　选择母版页

（6）在创建的内容页中，简单添加一些说明文字，代码如下：

```
<%--下面的@Page 语句中的 MasterPageFile 指定了页面所使用的母版页--%>
<%@  Page  Title=""  Language="C#"  MasterPageFile="~/Site1.Master"
AutoEventWireup="true" CodeBehind="WebForm1.aspx.cs" Inherits="SchoolWeb.
WebForm1" %>
<%--下面这个Content控件对应母版页中位于head标记中的ContentPlaceHolder控件--%>
```

```
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<%--下面这个Content控件对应母版页中位于body标记中的ContentPlaceHolder控件--%>
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
<h1>欢迎来到滨州学院</h1>
<h3>这是内容页里的内容</h3>
</asp:Content>
```

（7）在浏览器中查看内容页，运行界面如图 1.11 所示。



图 1.11　内容页运行界面

1.2.3　将现有页面转换为母版页或内容页

在网站或者 Web 应用程序开发过程中，有时需要把一个已经存在的普通 ASP.NET 页面修改为母版页或者内容页。把一个现有页面做成母版页的主要作用是可以复用此页面中的元素，而把现有页面做成内容页可以使此页面具有与母版相同的布局，从而实现网站风格的标准化和一致性。这两种转换都是通过修改页面的 HTML 代码（即.master 或者.aspx 文件里的代码）实现的。本节将介绍如何实现这两种转换。

仔细观察母版页、内容页与普通页面的 HTML 代码可以发现，三者第一行代码是不一样的。下面分别列出了普通页面、母版页、内容页的第一行代码。

```
//普通页面 OrdinaryPage.aspx 第一行代码
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="OrdinaryPage.
aspx.cs" Inherits="MasterPageSample.OrdinaryPage" %>
//母版页 MyMaster.master 第一行代码
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="MyMaster.
master.cs" Inherits="MasterPageSample.MyMaster" %>
//内容页 ContentPage.aspx 第一行代码
<%@ Page Title="" Language="C#" MasterPageFile= "~/MyMaster.Master"
```

```
AutoEventWireup ="true" CodeBehind= "ContentPage.aspx.cs" Inherits=
"MasterPageSample.ContentPage" %>
```

从上述代码可以看出，母版页的 master 文件以<%@Master 开头，普通页面和内容页面的 aspx 文件都是以<%@Page 开头，但是内容页面有一个 MasterPageFile 标记，表示此内容页面所使用的是母版页，而普通的 aspx 页面没有这个标记。从这第一行代码就可以判断出来一个页面是母版页、内容页还是普通页面。

再注意查看三个页面文件的其余内容可以发现，母版页中通常有两个内容占位符 ContentPlacerHolder，每一个占位符都有一个唯一 ID，正是这两个内容占位符为内容页面提供了定制页面的机制。下面是一个空白的母版页的代码（仅包含两个内容占位符）。

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="MyMaster.
master.cs" Inherits="MasterPageSample.MyMaster" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title></title>
   <!--这是第一个内容占位符，出现在 HTML 文档 head 部分-->
   <asp:ContentPlaceHolder ID="head" runat="server">
   </asp:ContentPlaceHolder>
</head>
<body>
   <form id="form1" runat="server">
   <div>
    <!--这是第二个内容占位符，出现在 HTML 文档 body 部分-->
      <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
      </asp:ContentPlaceHolder>
   </div>
   </form>
</body>
</html>
```

提示： 一个母版页并非固定只能包含两个内容占位符，而是可以包含一到多个占位符。这些占位符可以放在 HTML 文档的任意部分。开发人员可以根据需要灵活确定内容占位符的数量及位置。

ASP.NET 的内容页是在母版页的基础上新增加的内容。在内容页面中，没有 HTML、BODY、HEAD 等标记，这些标记在一个 HTML 文档里只出现一次，已经包含在母版页里了。内容页通常包含与母版页相对应的内容（Content）控件，一个内容控件对应于母版页里一个内容占位符控件，这个对应关系是由内容控件的 ContentPlaceHolderID 属性确定的。当显示一个内容页时，Content 控件中的内容就会插入到母版页中对应的 ContentPlaceHolder 控件所在的位置，从而组成一个完整的页面。下面是一个空白内容页的代码。

```
<%@ Page Title="" Language="C#" MasterPageFile="~/MyMaster.Master"
AutoEventWireup="true" CodeBehind="ContentPage.aspx.cs" Inherits=
"MasterPageSample.ContentPage" %>
<!--第一个内容控件，对应于母版里的第一个内容占位符（id 为 head）-->
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<!--第二个内容控件，对应于母版里的第二个内容占位符（id 为 ContentPlaceHolder1）-->
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
</asp:Content>
```

提示：母版页中包含 ContentPlaceHolder 控件，内容页包含 Content 控件。内容页中的 Content 控件与母版页中的 ContentPlaceHolder 控件一一对应，这种对应关系通过 Conent 控件的 ContentPlaceHolderID 属性确定。

前面分析了母版页、内容页中 HTML 代码的含义及两者对应关系，根据这些知识，可以容易地把一个普通 ASPX 页面转换成母版页、内容页，或者做相反的转换。下面通过两个例子做具体说明。

【例 1-6】 从普通页面提取母版页。

本例演示如何基于一个现有的普通 ASPX 页面提取出母版页。

（1）创建一个 ASP.NET Web 应用程序或者网站。

（2）在网站中添加一个普通 ASP.NET 页面，重命名为 OrdinaryPage.aspx。在页面中加入如下 HTML 代码。

```
<body>
<form id="form1" runat="server">
  <div>                                          <%--页面最外层 div--%>
  <div>                                          <%--页面顶部 div--%>
  <h3>这里放页面顶部的 flash，大横幅广告等</h3>
  <h3>这里放置页面部分链接</h3>
  </div>
  <div style=" border:solid 2px silver; height:200px;"> <h3>这里是页面的具
  体内容</h3></div>
  <div>这是页脚</div>
  </div>
</form>
</body>
```

分析上述页面的代码，可以看到大体可以分为三部分，页面顶部的 flash 和链接是一部分，页面中间的正文内容是一部分，页脚内容是一部分。这三部分内容中，页头和页脚内容对于所有页面是相同的。为了优化网站的设计，可以把页顶和页脚放到母版页中。

（3）在项目中添加一个母版页，重命名为 MyMaster.master。修改母版页的 body 部分，在内容占位符的前面加入页头，在内容占位符之后加入页脚。修改后的母版页代码如下：

```
<body>
<form id="form1" runat="server">
 <div>                                       <%--页面最外层 div--%>
  <div>                                      <%--页面顶部 div--%>
 <h3>这里放页面顶部的 flash，大横幅广告等</h3>
 <h3>这里放置页面部分链接</h3>
 </div>
      <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
      </asp:ContentPlaceHolder>              <%--母版页的内容占位符控件--%>
   <div>这是页脚</div>                       <%--页脚 div--%>
   </div>
</form>
</body>
```

【例 1-7】 将普通页面转换为内容页。

本例演示如何将一个现有的普通 ASPX 页面转换为一个内容页，即使其建立在一个母版页的基础上。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加一个新页面 OrdinaryPage.aspx，修改页面代码如下：

```
<head id="Head1" runat="server">
    <title>这是我的页面</title>
<script type="text/javascript"> alert('这是一段 javascript 代码'); </script>
</head>
<body>
    <form id="form1" runat="server">
  <div style=" border:solid 2px silver; height:200px;"> <h3>这是页面的内容
</h3></div>
    </form>
</body>
</html>
```

从上述代码可以看出，这是一个独立的 ASPX 页面，页面标题为“这是我的页面”，页面中一行文字。如果要修改此页面的外观与其他页面一致，给此页面加上与其他页面一样的页头、页脚，那么最合理也是简单的方法就是使用母版页，让此页面成为母版页的一个内容页面。

（3）把例 1-6 中的母版页 MyMaster.master 复制到本项目中。为节省篇幅，此处不重复列出母版页的代码。

（4）修改 OrdinaryPage.aspx 文件，在第一行代码<%@Page 的后面添加 MasterPageFile= “~/MyMaster.Master”，表示此页面是基于 MyMaster 母版页创建的一个内容页，再添加“Title="这是我的网站”，表示内容页的标题是“这是我的网站”。修改后的代码如下：

```
<%@ Page MasterPageFile="~/MyMaster.Master" Title="这是我的网站"  Language=
"C#" AutoEventWireup="true" CodeBehind="OrdinaryPage.aspx.cs" Inherits=
"MasterPageSample.OrdinaryPage" %>
```

（5）由于内容页中没有 HTML、BODY、HEAD 等标记，所以在 OrdinaryPage.aspx 中删除这些标记。注意删除的时候不能删除页面内容。删除这些标记后的页面代码如下：

```
<%@ Page MasterPageFile="~/MyMaster.Master" Title="这是我的网站"  Language=
"C#" AutoEventWireup="true" CodeBehind="OrdinaryPage.aspx.cs" Inherits=
"MasterPageSample.OrdinaryPage" %>
<script  type="text/javascript">     alert('这是一段 javascript 代码');
</script>
<div  style="  border:solid  2px  silver;  height:200px;">  <h3>这是页面的内容
</h3></div>
```

（6）在内容页 OrdinaryPage 中添加 Content 控件。如前所述，内容页的 Content 控件与母版页的 ContentPlaceHolder 控件是一一对应的，所以，根据母版页中 ContentPlaceHolder 控件的多少及其 ID 可以确定内容页中 Content 控件的个数和属性。代码如下：

```
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<asp:Content   ID="Content2"   ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
</asp:Content>
```

（7）把原有内容页中的网页内容移动到相应的 Content 控件中。在本例中，把原有页面 head 标记中的内容移动到 ContentPlaceHolderID 为 head 的 Content 控件中，而把页面正

文内容移动到 ContentPlaceHolderID 为 ContentPlaceHolder1 的 Content 控件中。完成后整个页面代码如下：

```
<%@ Page MasterPageFile="~/MyMaster.Master" Title="这是我的网站" Language=
"C#" AutoEventWireup="true" CodeBehind="OrdinaryPage.aspx.cs" Inherits=
"MasterPageSample.OrdinaryPage" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<script type="text/javascript">    alert('这是一段 javascript 代码');
</script>
</asp:Content>
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
<div style=" border:solid 2px silver; height:200px;"> <h3>这是页面的内容
</h3></div>
</asp:Content>
```

1.2.4　嵌套母版页

基于母版页不但可以创建出内容页，而且还可以创建出其他母版页。这种基于母版页创建的母版页称为嵌套母版页。母版页可以嵌套多层，但是从实用的角度来说，母版页嵌套一般不超过两层。

母版页嵌套是一种非常实用的机制。例如，某网站有许多页面，这些页面按照外观布局大致可以分为两大类，第一类页面由纵向排列的页头、内容、页脚三部分内容组成，第二类页面与第一类页面不同之处在于中间的内容部分又拆分成为两部分：左侧导航栏和页面内容。这两大类页面的页头和页脚都是相同的。利用母版页嵌套就很容易设计出合理的母版和页面。下面通过一个例子具体说明这种设计方法。

【例 1-8】 嵌套母版页。

本例演示嵌套母版页的使用。网站中有两个母版页，第一个母版页包含页头、内容占位符和页脚。第二个母版页是一个嵌套母版页，在第一个母版页的基础上，把第一个母版页的内容占位符拆分成了左右两部分，左边是导航栏，右边是内容占位符。

利用嵌套母版页创建两类既有联系又有区别的页面。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加一个母版页 Master1.master，这个母版页包含页头、内容占位符和页脚三部分。为了让各个部分边界更加清晰，页面中使用了一个 CSS 样式。页面代码如下：

```
<head runat="server">
   <style type="text/css">                    <%--页面中用到的 CSS 样式--%>
   div { border: solid 2px silver; margin:10px;}
   </style>
   <asp:ContentPlaceHolder ID="head" runat="server">
                                              <%--head 中的占位符--%>
   </asp:ContentPlaceHolder>
</head>
<body>
   <form id="form1" runat="server">
   <div>
   <div><h3>第一个母版页 HEADER</h3></div>
      <%--页面 body 中的内容占位符控件--%>
      <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
```

```
        </asp:ContentPlaceHolder>
        <div>
        第一个母版页 FOOTER
        </div>
    </div>
    </form>
</body>
</html>
```

Master1.master 页面在设计状态下的外观如图 1.12 所示。

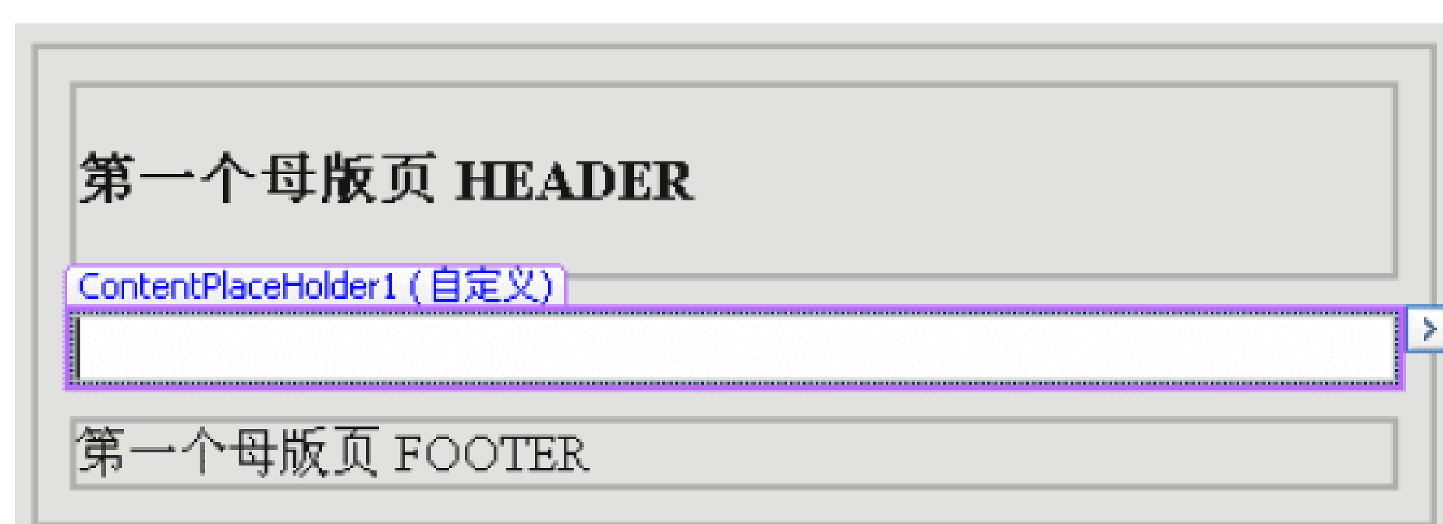


图 1.12　第一个母版页

（3）从 Visual Studio 菜单中选择“项目”|“添加新项”命令，在打开的“添加新项”对话框中选择“嵌套的母版页”，将新的母版页命名为 Master2.master。在随后出现的“选择母版页”对话框中，选择第（2）步所创建的 Master1。所生成的嵌套母版页代码如下：

```
<%@ Master Language="C#" MasterPageFile="~/NestedMaster/Master1.Master"
AutoEventWireup="true"   CodeBehind=   "NestedMasterPage1.   master.cs"
Inherits= "MasterPageSample.NestedMaster. Master2" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<asp:Content   ID="Content2"   ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
</asp:Content>
```

从上面这段代码可以看出，自动生成的嵌套母版页的代码几乎与自动生成的内容代码是相同的，也是包含两个 Content 控件，这两个 Content 控件对应于母版 Master1 中的两个 ContentPlaceHolder。但是嵌套母版页的第一行代码与内容页不同。母版页以“<%@Master”开头，而内容页以“<%@Page”开头，另外，内容页有一个 Title 属性，而母版页则没有这个属性。

现在这个母版页其实还算不上一个真正的母版页，因为还不包含 ContentPlaceHolder 控件。下面的步骤就来放置 ContentPlaceHolder 控件。

（4）在 Master2.master 母版页中的第一个 Content 控件中放置一个 ContentPlaceHolder 控件，代码如下：

```
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <asp:ContentPlaceHolder ID="head2" runat="server">
    </asp:ContentPlaceHolder>
</asp:Content>
```

要注意区分上面这段代码中的 Content、ContentPlaceHolder 及其关系。

（5）在 Master2.master 母版页中的第二个 Content 控件中放置两个 div 标记，第一个 div 标记中包含左侧导航栏，第二个 div 中包含一个 ConentPlaceHolder 控件，为内容页留出存放页面内容的空间。

```
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
   <div style="float:left;">
<h3>第二个母版页<br />左侧导航栏</h3>
</div>
   <div style="float:left; ">
      <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
      </asp:ContentPlaceHolder>
   </div>
</asp:Content>
```

Master2.master 页面在设计状态下的外观如图 1.13 所示。

（6）在项目中添加一个基于母版页 Master2 的内容页 Master2Page.aspx，并在其中写入些文字，代码如下：

```
<%@ Page Title="" Language="C#" MasterPageFile="~/NestedMaster/Master2.
master" AutoEventWireup="true" CodeBehind="Master2Page.aspx.cs" Inherits=
"MasterPageSample.NestedMaster.Master2Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head2" runat="server">
</asp:Content>
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
   <h3>内容页面 2 具体内容</h3>
</asp:Content>
```

（7）在浏览器中查看 Master2Page.aspx 页面，效果如图 1.14 所示。

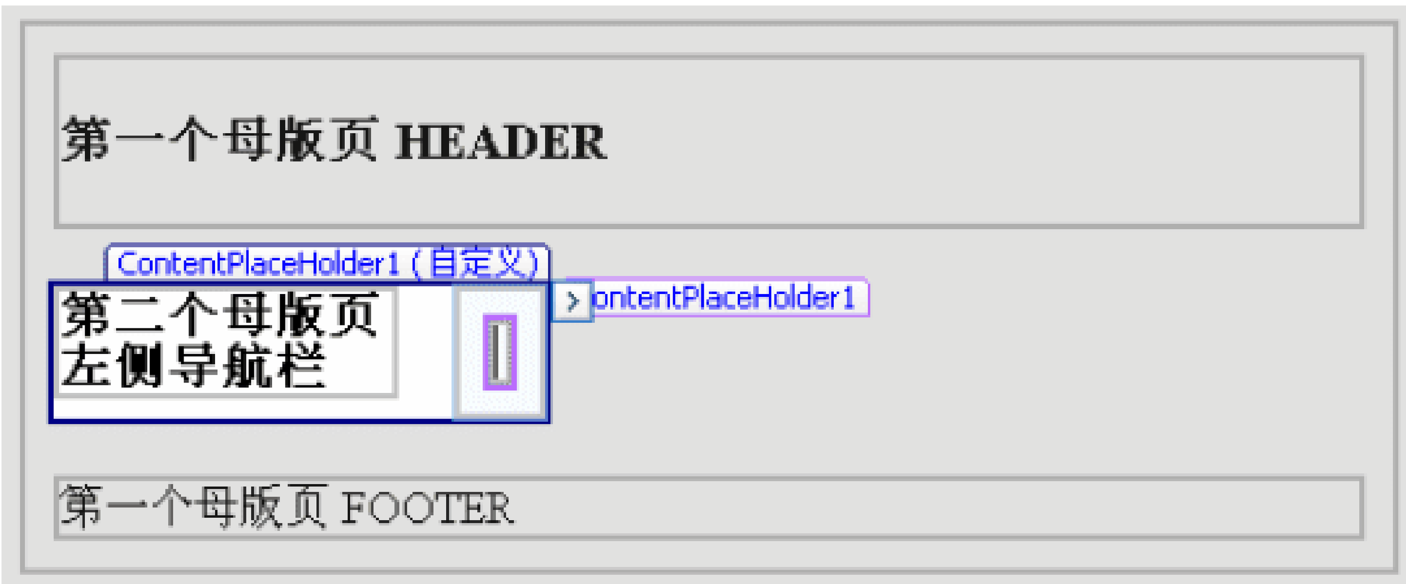


图 1.13　嵌套母版页

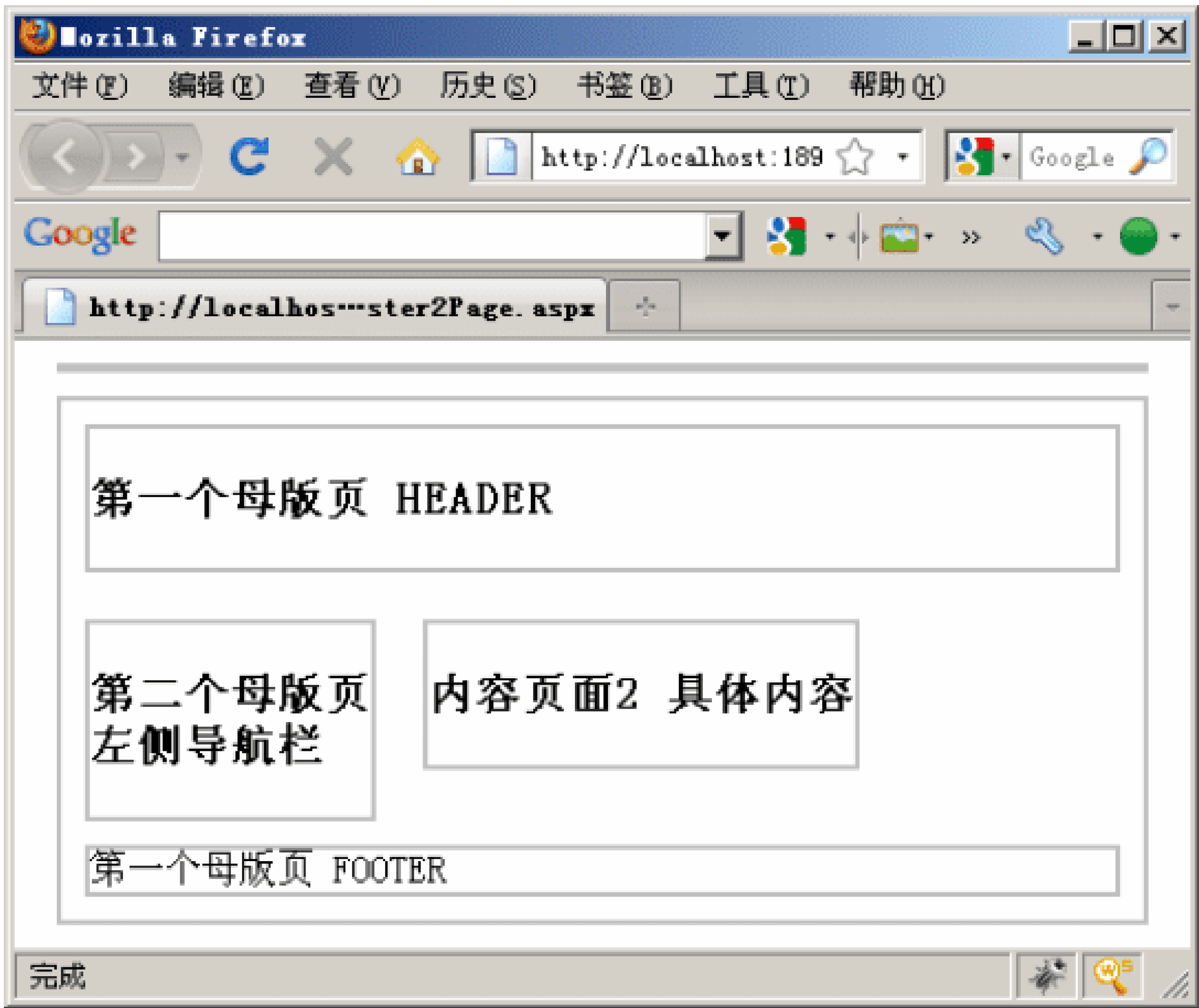


图 1.14　内容页外观效果

1.2.5　从内容页访问母版页控件

在 ASP.NET 开发环境下，母版页与内容页是两个不同的页面，存储在两个不同的文件中，内容页不能直接访问母版页的控件。但是在实际应用中，却经常需要这种访问，这就需要通过一些编程技巧来实现。

ASP.NET 页面是 System.Web.UI.Page 类的派生类，母版页是 System.Web.UI.MasterPage 类的派生类。Page 类有一个 Master 属性，表示此页面的母版页，代码如下：

```
public MasterPage Master { get; }
```

通过 Page 类的 Master 属性，就可以访问到页面的母版页。而 MasterPage 类继承自 UserControl 类，因此调用其 FindControl()方法就可以得到母版页上的某个控件。

【例 1-9】 访问母版页控件（方法 1）。

许多网站的页面上都有一个“当前位置”的标识，用于说明当前正在访问的页面名称。由于这个标识存在于多个页面，所以可以把这个标识放在母版页中。但是对于各个不同页面来说，这个标识是不一样的，需要在内容页中修改这个标识的内容。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加一个母版页 Site1，并在母版页上放置一个 ID 为 currentPosition 的 Label，用于显示当前位置。页面主要代码如下：

```
<body>
    <form id="form1" runat="server">
    <div>
    <div>
    <div><h3>页面顶部固定内容</h3></div>
    <div>当前位置:
        <asp:Label ID="currentPosition" runat="server" Text="你的位置">
        </asp:Label> </div>
    </div>
        <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
        </asp:ContentPlaceHolder>
        <div>页脚固定内容</div>
    </div>
    </form>
</body>
```

（3）在项目中添加一个基于母版页 Site1 的内容页 HomePage.aspx。在内容页的 Load 事件中，编写代码修改母版页中 Label 控件的文本。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        //得到母版页上用于标识当前位置的 Label 控件
        Label position = this.Master.FindControl("currentPosition") as
        Label;
        //如果找到了这个 Label 控件，同时设置其文本
        if ( position!=null)
            position.Text = "<a href='HomePage.aspx'>首页</a>";
    }
}
```

（4）运行 HomePage.aspx，运行界面如图 1.15 所示。

例 1-9 演示了通过 FindControl 访问母版页控件的方法。这种方法的优点是简单、适应性强，只需要编写少量代码，就可以用于访问任何类型控件。但这种方法也存在以下几个缺点。

- ❑ 如果母版页里控件很多，很难记住其中控件的 ID。
- ❑ 在调用 FindControl 时，如果把控件的 ID 写错了，Visual Studio 不能给出错误提示，只有当程序运行时才会产生错误。
- ❑ FindControl 方法返回 Control 类型而非具体类型（如 Label），需要进行强制类型转换。

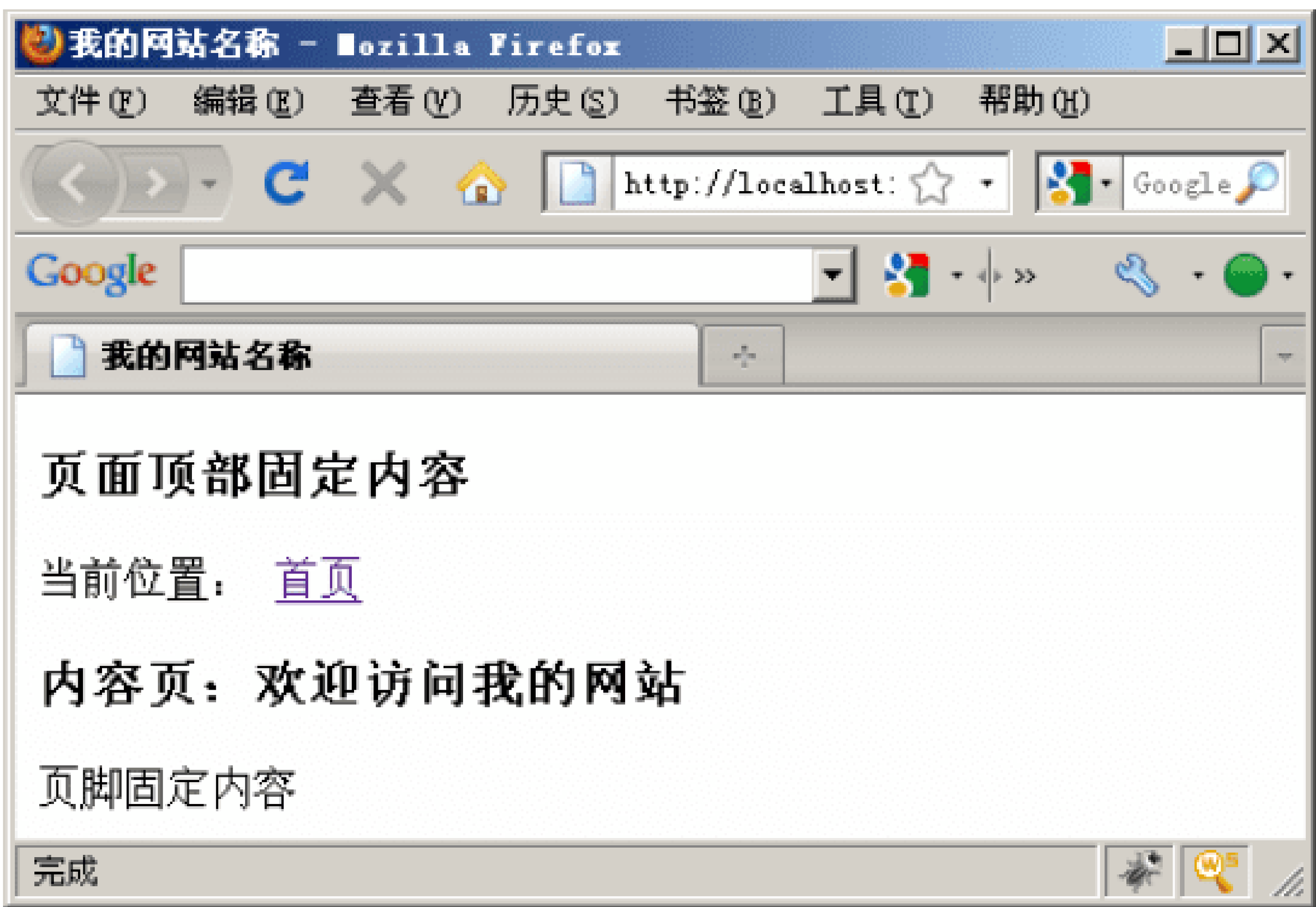


图 1.15　例 1-9 运行界面

- 从面向对象的角度来讲，这种方式从一个类（内容页）访问另一个类（母版页）的私有成员，破坏了母版页的封装。

针对上述缺点，可以用另外一种方式访问母版页控件。这种方式的思路是在母版页中声明一个属性，内容页通过这个属性来访问母版页上的控件。这样，只有母版页本身才与其中的控件打交道，内容页不需要知道母版页上控件的 ID，而且属性是强类型的，从而很好地解决了以上几个问题。

【例 1-10】 访问母版页控件（方法 2）。

（1）创建一个 ASP.NET 应用程序，添加一个母版页 Site1.master，其 HTML 代码与例 1-9 完全相同。

（2）在 Site1.master.cs 文件中，为母版页添加一个属性 myPosition，用于控制母版页 Label 上的文本。代码如下：

```
//将母版页中标记当前位置的 Label 的文本封装成为一个属性以方便访问
public string MyPosition
{
    get { return currentPosition.Text; }
    set { currentPosition.Text = value; }
}
```

（3）在项目中添加一个内容页 HomePage.aspx，并在页面的 Load 事件中设置当前位置。代码如下：

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        Site1 master = this.Master as Site1;          //获得母版页
        master.MyPosition = "首页";                    //设置母版上的当前位置
    }
}
```

1.3　主　　题

ASP.NET 的主题（Theme）是一种可以集中控制网页控件外观的机制。通过应用主题，

可以使网站中的控件具有一致的外观，例如，让所有的 Button 都具有相同的边框、背景色、字体等。ASP.NET 的主题与 HTML 中的 CSS 既有联系又有区别，两者经常配合使用。

1.3.1　创建和使用主题

与 ASP.NET 主题密切相关的一个概念是外观（Skin）。一个主题确定了一种页面风格，而一个外观则定义了一种控件的样式。Skin 的意思是皮肤，这个单词很明确地表明了 ASP.NET Skin 的作用，就是定义控件呈现出来的样子。通过将主题和外观应用于网站或者网页，就可以定制该范围内控件的样式。

在 ASP.NET 中，一个主题对应于一个特殊的文件夹，而一个外观则通常对应于某主题文件夹下的一个文件。一个主题可以包含 0 到多个外观文件，这些外观文件定义了各种控件在该主题下所呈现出来的样式。

下面通过一个例子介绍如何使用主题和外观。

【例 1-11】 应用主题和外观。

本例演示如何创建主题，如何添加外观，以及如何将所创建的主题和外观应用于网站和页面。

（1）创建一个 ASP.NET Web 应用程序，命名为 ThemeSample。

（2）在 Visual Studio 解决方案资源管理器中，右击项目名称，从弹出的快捷菜单中选择“添加”|“添加 ASP.NET 文件夹”|“主题”命令，如图 1.16 所示。



图 1.16　添加主题文件夹

（3）添加主题后，会在项目中增加两个文件夹：一个是 App_Theme，另外一个是“主题 1”文件夹。其中 App_Theme 是一个 ASP.NET 系统文件夹，用于存放网站中的主题。这个文件夹只有当第一次添加主题时才会创建。另外一个“主题 1”文件夹就是主题所对应的文件夹，文件夹的名称就是主题的名称。把“主题 1”重合名为 GrassGreen。

（4）在 GrassGreen 主题文件夹上右击，从弹出的快捷菜单中选择“添加”|“新建项”命令，在弹出的“添加新项”对话框中选择“外观文件”，把文件重命名为 Sample.skin，单击“添加”按钮。

（5）删除 Sample.skin 文件中自动生成的内容，然后添加以下代码：

```
<asp:Button runat="server" BackColor="lightgreen"  BorderStyle="Solid"
BorderColor= "DarkGreen"/>
```

注意：在外观文件中不能包含除控件标记以外的其他文字内容，包括不能包含注释，否则页面运行时会提示分析器错误。

上面的代码设置了 Button 控件的外观：浅绿色背景，深绿色实线边框。可以看出，这种设置外观的语法与 ASP.NET 页面中设置控件的语法相同，而与 HTML 中 CSS 的语法不同。

提示：外观文件中通常包含很多属性，而且不能直接看到这些属性对应的效果，这给外观文件代码编写造成不小的困难。可以先在 ASP.NET 页面中以所见即所得方式设置好控件外观属性，然后再复制到外观文件中。注意在复制代码时要把控件 ID 删除。

（6）前面几个步骤创建了主题和一个外观文件，下面将使用该主题和外观。在项目中添加一个页面 ThemePage.aspx，在页面上放两个 Button 控件。此时按钮的外观仍然是默认的灰色按钮。如果要应用前一步骤中所创建的主题，那么应该修改一下 ThemePage.aspx 页面代码，在页面第一行的<%@ Page 后面添加一个 Theme 属性，代码如下：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="ThemePage.
aspx.cs" Inherits="ThemeSample.ThemePage" Theme="GrassGreen" %>
```

（7）在浏览器中查看 ThemePage 页面，可以看到两个按钮都变成了绿色背景和边框。

（8）在 ThemePage.aspx 页面上添加一个 HTML 的 Button，再次运行页面，可以看到 ASP.NET 的 Button 变成绿色，而 HTML 的 Button 并没有改变颜色。这是因为主题和外观仅对服务器端控件有效。

提示：ASP.NET 主题和外观仅对服务器端控件有效，对于 HTML 客户端控件无效。

上一个例子实现了主题、外观的创建和应用，仔细考虑这个例子，会发现两个问题。

（1）如果一个页面有多个 Button 控件，有的 Button 需要应用主题和外观，而有的则不需要，应该如何加以控制。

（2）一个网站中有许多页面，所有页面都需要应用主题。如果每个页面修改<%@Page 语句的话，工作量太大。而且将来如果要更改主题，那么修改的工作量也相当大。

这两个问题在 ASP.NET 中都有很好的解决方案，使用命名主题可以解决第一个问题，使用配置文件可以解决第二个问题。下面通过一个例子演示具体的代码实现。

【例 1-12】 命名外观和配置文件。

（1）创建一个 ASP.NET 应用程序。

（2）在项目中添加一个主题 SkyBlue。在主题中添加一个外观文件 Test.skin，代码如下：

```
<asp:Button SkinID="skin1" runat="server" BackColor="AliceBlue"
BorderStyle="Solid" BorderColor="Blue" Font-Bold="true" />
<asp:Button SkinID="skin2" runat="server" BackColor="Silver" BorderStyle=
"Solid" BorderColor="Blue" />
```

在上述外观文件代码中，每个控件都有一个 SkinID 属性，这个属性指定了此外观的名称，这种外观称为命名外观。在例 1-11 中所创建的不带 SkinID 的外观称为默认外观。如果页面使用了主题，则默认外观会自动应用到页面上的对应控件，而命名外观必须通过其

名称明确引用。

注意：在同一个主题中，同一种控件不允许有重名的命名外观，也不允许有多于一个默认外观。

Test.skin 文件中为 Button 控件定义了两个命名外观，其 ID 分别为 skin1 和 skin2。skin1 定义的 Button 样式为浅蓝色背景、蓝色实线边框、粗体字，skin2 定义的 Button 样式为银灰色背景、蓝色边框。

（3）在项目中添加一个页面 ThemePage2.aspx，为页面指定主题为 SkyBlue，页面中包含三个 Button 控件，页面关键代码如下：

```
<%@  Page  Language="C#"  AutoEventWireup="true"  CodeBehind="ThemePage2.
aspx.cs" Inherits="ThemeSample.sample2.ThemePage2" Theme="SkyBlue" %>
<body>
    <form id="form1" runat="server">
    <div>
     <asp:Button ID="Button1" runat="server" Text="测试按钮 1"/>
     <asp:Button ID="Button2" runat="server" Text="测试按钮 2"/>
     <asp:Button ID="Button3" runat="server" Text="测试按钮 3"/>
    </div>
    </form>
</body>
```

此时如果浏览 ThemePage2.aspx 页面，可以看到 3 个按钮都是默认样式，说明在 Test.skin 外观文件中定义的 2 个 Button 的命名外观并没有被应用。

（4）通过 Button 控件的 SkinID 属性为前 2 个按钮分别指定 skin1 和 skin2 外观，而第 3 个按钮不使用命名外观。代码如下：

```
<asp:Button ID="Button1" runat="server" Text="测试按钮 1"  SkinID="skin1" />
<asp:Button ID="Button2" runat="server" Text="测试按钮 2"  SkinID="skin2" />
<asp:Button ID="Button3" runat="server" Text="测试按钮 3" />
```

（5）在浏览器中查看 ThemePage2.aspx，可以看到 Button1 样式为浅蓝色背景、蓝色实线边框、粗体字，Button2 样式为银灰色背景、蓝色实线边框，说明两个按钮已经分别应用了 skin1 和 skin2 外观。ThemePage2 页面运行界面如图 1.17 所示。

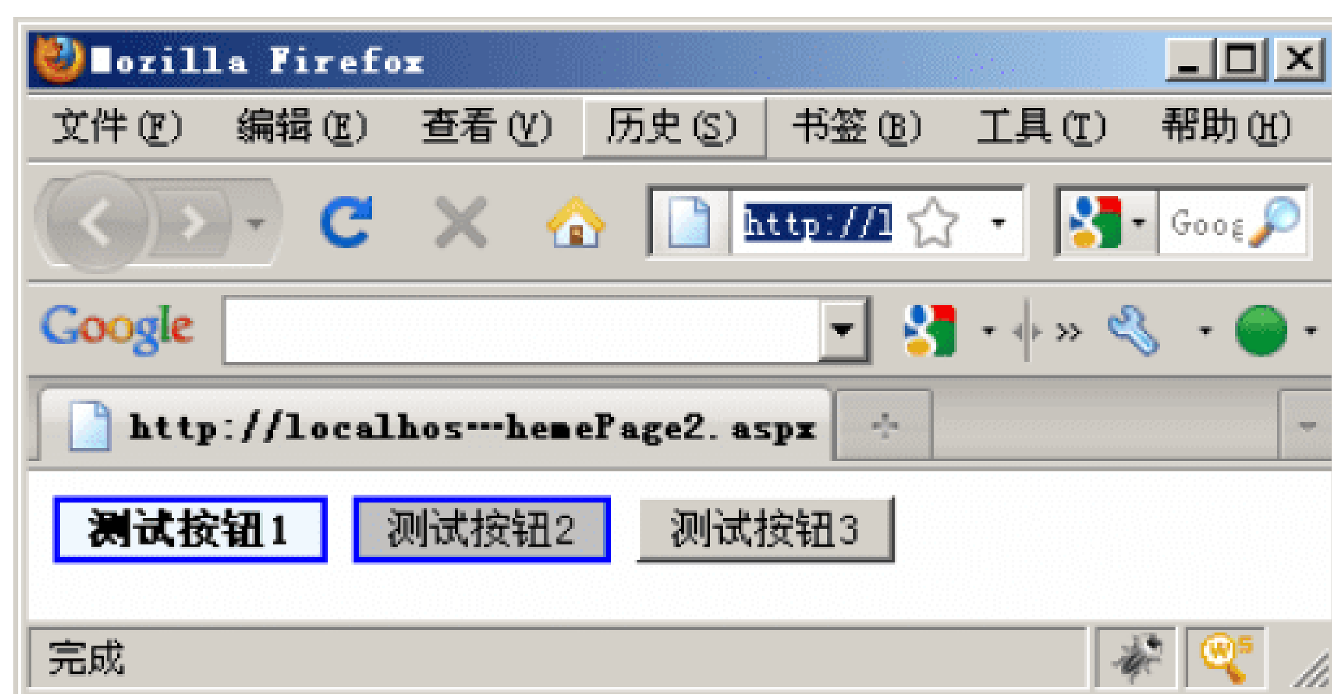


图 1.17　ThemePage2 运行界面

（6）通常网站中会有多个页面都需要使用主题。通过修改<%@Page 页面指令来使用主题显然是一种费时费力的笨办法，可以通过配置文件一次性将主题应用于所有页面。在本项目中，打开 Web 配置文件 web.config（如果文件不存在，则添加一个），在 system.web

结点中添加一个 pages 结点，并指定主题，代码如下：

```
<system.web>
  <pages theme="SkyBlue"/>
</system.web>
```

（7）打开 ThemePage2.aspx 代码，并去掉<%@Page 后面的 Theme 属性，重新编译运行该页面，可以看到 Button 控件的外观仍然有效。这是由于通过 web.config 中的配置已经把 SkyBlue 主题应用到了所有页面。如果再添加新的页面，也会自动应用 SkyBlue 主题。

1.3.2　主题与样式表

主题与样式表 CSS 都可用于控制页面外观。主题只能用于 ASP.NET 服务器控件，而 CSS 可用于任意 HTML 元素，包括 ASP.NET 服务器控件。由于主题是专门针对 ASP.NET 控件设计的，所以更能灵活、精确地设置 ASP.NET 控件的各种外观属性，其语法也接近 C#代码，易于为 C#程序员所掌握。在开发 Web 程序时把主题和 CSS 结合使用，可以实现二者优势互补，更好地设计出美观的页面。

要在 HTML 中引用一个 CSS 文件，通常使用以下语法，其中 MyCss.css 为要引用的 CSS 文件名。

```
<link href="MyCss.css" rel="stylesheet" type="text/css" />
```

当主题和 CSS 结合使用时，可以把 CSS 文件放到 ASP.NET 的主题文件夹中。如果页面应用了某主题，则主题文件夹中的所有 CSS 文件都自动被页面所包含，而不需要在页面中写 link 语句。

【例 1-13】 主题文件夹中的 CSS 文件。

本例演示主题文件夹中的 CSS 文件会自动被应用该主题的页面所引用。

（1）创建一个 ASP.NET 应用程序。

（2）在项目中添加主题文件夹，并添加一个名为 CssTheme 的主题。

（3）在 CssTheme 文件夹中添加一个 CSS 文件 MyCss.css，其中包含以下代码：

```
/*定义所有 input 元素默认样式为黄色背景，蓝色边框*/
input
{
    background-color:Yellow;                /*黄色背景色*/
    border:solid 1px BLUE;                  /*边框：实线、蓝色、1 像素宽*/
}
/*定义一个名为 button 的 CSS 类，银灰色背景，黑色边框*/
.button
{
    background-color:silver;                /*银灰色背景色*/
    border:solid 1px black;                 /*边框：实绩、黑色、1 像素宽*/
}
```

（4）在项目中添加一个页面 CssThemePage.aspx，设置页面的主题为 CssTheme，并在页面上放置一个 TextBox 和一个 Button 控件，关键代码如下：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="CssThemePage.
aspx.cs" Inherits="ThemeSample.CssTheme.CssThemePage" Theme="CssTheme" %>
```

```
<body>
   <form id="form1" runat="server">
   <div>
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
      <asp:Button ID="Button1" runat="server" Text="Button" CssClass=
      "button" />
   </div>
   </form>
</body>
</html>
```

（5）运行此页面，可以看到页面上的 TextBox 控件和 Button 控件都应用了相应的 CSS 样式。从浏览器菜单中选择“查看源码”命令，可以看到所生成页面的 HTML 代码中包含以下内容：

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
<title></title>
<link    href="App_Themes/CssTheme/MyCss.css"    type="text/css"    rel=
"stylesheet" />
</head>
```

其中的 link 一行代码引用了 CssTheme 主题文件夹里的 CSS 文件。这也证实了包含在页面主题文件夹里的 CSS 文件会被自动引用。

应用主题可以批量指定 ASP.NET 服务器控件的样式，如果要单独设置某一个控件的样式，通常是在 ASPX 页面中为控件指定相应属性，例如下面的代码把 Button 背景色设置为浅绿色。

```
<asp:Button  ID="Button1"  runat="server"  Text=" 测 试 按 钮  1"  BackColor=
"LightGreen"  />
```

如果对于同一控件，既应用了主题和外观，又在 ASPX 页面中为控件定义了内联样式，那么控件在显示时会出现什么结果呢？如果出现这种情况，ASP.NET 会把这主题的样式和页面中的内联样式合并起来，并按照一定的优先级解决两者的冲突。如果在配置文件或者页面<%@Page 指定中使用了 Theme 关键字，那么这种主题的优先级则高于页面中内联样式。如果要想让页面中的内联样式优先级高于主题样式，则在应用主题时应该使用 StyleSheetTheme 关键字。

提示： Theme、控件内联样式、StyleSheetTheme 三者的优先级依次降低。

【例 1-14】 Theme 和 StyleSheetTheme。

本例演示 Theme、StyleSheetTheme、控件内联样式的合并及各自的优先级。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加一个 GrassGreen 的主题，并在该主题中添加一个 skin 文件。skin 文件为 Button 控件定义了默认外观（浅绿色背景色、深绿色边框），代码如下：

```
<asp:Button  runat="server" BackColor="lightgreen"  BorderStyle="Solid"
BorderColor="DarkGreen"/>
```

（3）在项目中添加一个页面 Page1.aspx，设置该页面主题为 GrassGreen，在页面上放置一个 Button，并设置 Button 的内联样式。页面关键代码如下：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Page1.aspx.cs"
Inherits="ThemeSample.StyleSheetTheme.Page1" Theme="GrassGreen"%>
<body>
    <form id="form1" runat="server">
   <div>
   <asp:Button ID="Button1" runat="server" Text="按钮一" BackColor="White"
Font-Bold="true" />
   </div>
    </form>
</body>
</html>
```

（4）在浏览器中查看 Page1 页面，可以看到 Button 样式为浅绿色背景、绿色实线边框、粗体字。这个样式是主题中定义的外观与 Button 内联样式综合作用的结果。在生成页面时，ASP.NET 首先把主题外观与内联样式合并，得到如下的样式：

```
BackColor="lightgreen"    BorderStyle="Solid"    BorderColor="DarkGreen"
BackColor="White" Font-Bold="true"
```

这个样式中包含两个 BackColor，第一个 BackColor 来源于主题外观，第二个 BackColor 来源于内联样式。这两个 BackColor 冲突，按照优先级来处理，由于 Theme 优先级高于内联样式，所以最终 Button 的 BackColor 为主题中定义的 LightGreen。

（5）把 Page1.aspx 页面的<%@Page 页面指令中 Theme 关键字改为 StyleSheetTheme，修改后的代码如下：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Page1.aspx.cs"
Inherits="ThemeSample.StyleSheetTheme.Page1" StyleSheetTheme="GrassGreen"%>
```

（6）再次浏览 Page1.aspx 页面，可以看到 Button 的样式为白色背景、绿色实线边框、粗体字。这个样式同样也是主题与内联样式综合作用的结果。在生成页面时，ASP.NET 首先把两种样式合并，然后按照优先级解决合并后得到的样式中的冲突。由于内联样式优先级高于 StyleSheetTheme，所以最终 Button 的背景色为内联样式中定义的白色。

1.3.3　动态修改主题

在有的网站中，允许用户根据自己喜好更换网站的皮肤，这个功能利用 ASP.NET 的主题机制能够很方便的实现。在 C#代码中可以通过 Page 类的 Theme 属性改变页面的主题，代码如下：

```
Page.Theme="要设置的主题名称";
```

上面代码只有一行，看似简单，但是要想在实际应用中发挥作用，还需要考虑多个因素。例如，让用户修改主题的一种最直观的思路是在页面上用一个下拉列表框列出主题，让用户选择其中一个，然后单击一个按钮，在按钮的 Click 事件中根据用户选择的主题进行修改。但是这种思路并不可行，因为根据对 ASP.NET 页面生命周期的分析，当处理 Button 的 Click 事件时，页面主题已经应用完毕，此时不可以再修改主题。

主题必须在页面初始化以前（通常是在页面的 PreInit）的事件中进行修改，而此时控件的状态尚不可用，也就无法获得 DropDownList 中所选择的值。这样就形成了一种矛盾：

在页面生命周期的早期阶段（如 PreInit 事件中），可以修改页面主题，但是无法读取控件状态；在页面生命周期的后期阶段（如 Button 的 Click 事件中），可以读到控件状态，但是却无法修改页面主题。要想解决这个矛盾，需要一定的编程技巧。下面通过一个例子具体说明。

【例 1-15】 动态修改主题。

本例演示如何以编程方式动态改变页面主题。

（1）创建一个 ASP.NET 应用程序。

（2）在项目添加一个主题，分别命名为 GrassGreen 和 SkyBlue。

（3）在 GrassGreen 主题下添加一个外观文件，此文件定义了 DropDownList 和 Button 的默认外观，代码如下：

```
<asp:Button  runat="server" BackColor="lightgreen"  BorderStyle="Solid"
BorderColor="DarkGreen"/>
<asp:DropDownList runat="server" BackColor="#aaffcc" BorderStyle="Solid"
BorderColor="Blue"/>
```

（4）在 SkyBlue 主题下添加一个外观文件，此文件也定义了 DropDownList 和 Button 两种控件的默认外观，代码如下：

```
<asp:Button  runat="server"  BackColor="AliceBlue"  BorderStyle="Solid"
BorderColor="Blue" />
<asp:DropDownList  runat="server"  BackColor="AliceBlue"  BorderStyle=
"Solid" BorderColor="Blue"/>
```

（5）在项目中添加一个 ASP.NET 页面 Page1.aspx，页面上放置一个 DropDownList 控件，列出可供选择的主题，还有一个 Button 控件，用以改变主题。代码如下：

```
请选择一个主题：<asp:DropDownList ID="DropDownList1" runat="server">
<asp:ListItem Value="GrassGreen">草绿色</asp:ListItem>
<asp:ListItem Value="SkyBlue">天蓝色</asp:ListItem>
</asp:DropDownList>
<asp:Button ID="Button1" runat="server" Text="更改主题"  onclick="Button1
Click" />
```

（6）为“更改主题”按钮的 Click 事件编写代码，以修改页面主题。代码如下：

```
protected void Button1 Click(object sender, EventArgs e)
{
   this.Theme = DropDownList1.SelectedValue;
}
```

（7）运行 Page1.asxp 页面，选择一个主题，并单击按钮修改主题。可以看到程序运行的结果并不像所期望的那样修改了页面的主题，而是出现以下错误提示：

```
"Theme"属性只能在"Page_PreInit"事件之中或之前设置。
```

这个错误提示证明了前面关于页面生命周期和主题设置的讨论，在服务器控件事件处理程序中不能修改页面主题，这种思路行不通。根据提示信息，很自然想到可以尝试在页面的 PreInit 事件中修改主题，下面来验证这种思路。

（8）为页面的 PreInit 添加事件处理程序，并在该事件处理程序中设置页面主题。代码如下：

```
protected void Page_Load(object sender, EventArgs e)
{
    this.PreInit += new EventHandler(Page PreInit);
}
void Page PreInit(object sender, EventArgs e)
{
    this.Theme = string s=DropDownList1.SelectedValue;
}
```

（9）运行 Page1.aspx 页面，选择一个主题并单击“更改主题”按钮。从运行结果可以看到，这个程序不能改变页面的主题。这是由于在页面的 PreInit 事件中，控件数据没有加载完毕，不知道 DropDownList 控件选中了哪一项。

前面尝试的两种方式都没有达到想要的效果，解决这个问题需要一种新的思路。可以考虑把用户所选择的主题作为一个 QueryString 参数传递，然后在 Page 类的 PreInit 事件中读取此参数并修改主题。

（10）在 Page1.aspx 页面中添加一个客户端按钮，并为此按钮编写 JavaScript 脚本。代码如下：

```
<head runat="server">
    <title>动态修改主题</title>
    <script type="text/javascript" language="javascript">
        function changeTheme() {
            //获得当前所选中的主题
            var theme = document.getElementById('<%=DropDownList1.Client
            ID%>').value;
            //重新加载当前页面，并通过 QueryString 传递所选中的主题
            window.location = "page1.aspx?theme=" + theme;
        }
    </script>
</head>
<body>
...
<input type="button"  value="更改主题（客户端 button）" onclick="change
Theme()" />
```

（11）修改 Page 类 PreInit 事件的代码，从 QueryString 中读取新的主题。代码如下：

```
void Page_PreInit(object sender, EventArgs e)
{
    this.Theme = Request.QueryString["theme"]??"GrassGreen";
}
```

（12）运行此页面，并修改主题，程序运行正常。运行界面如图 1.18 所示。

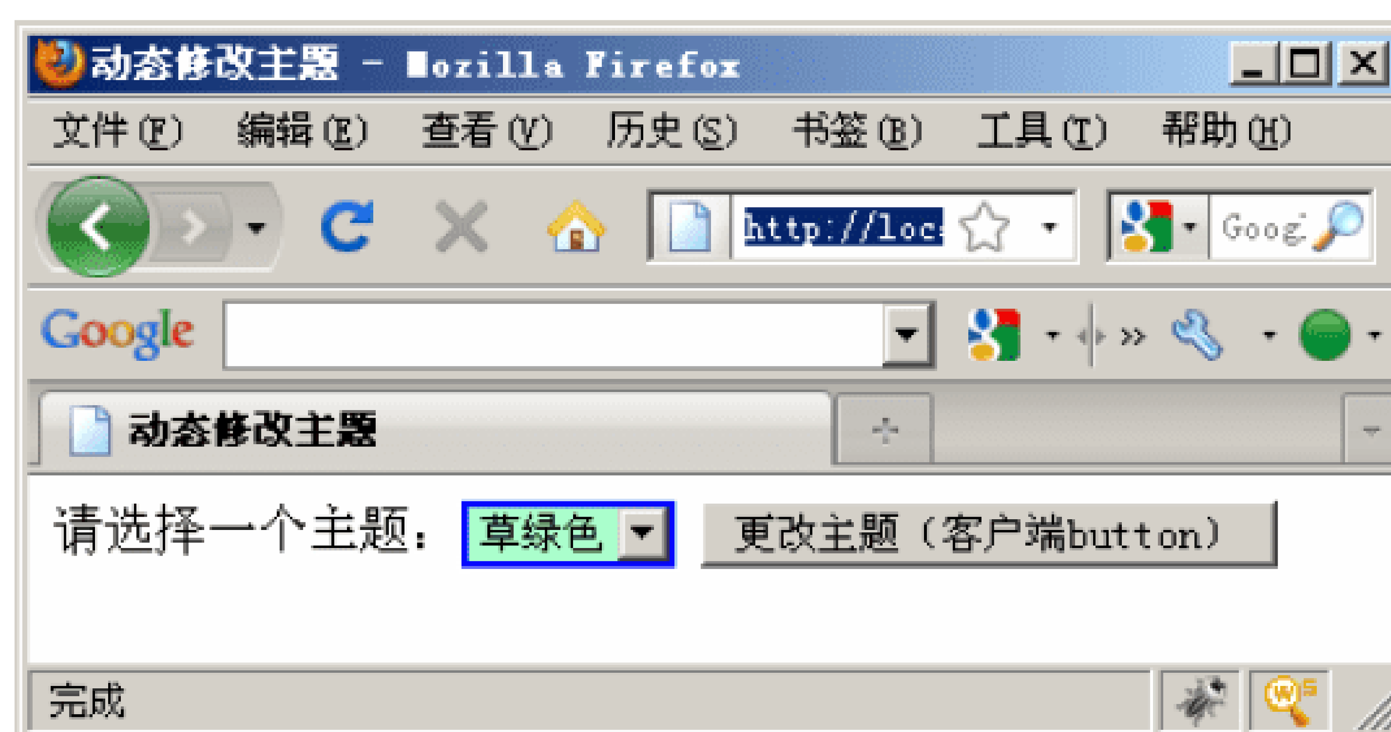


图 1.18　动态修改主题示例运行界面

1.4　Web 服务

Web 服务（Web Services）是一种面向服务的架构技术，通过标准的 Web 协议提供服务，目的是保证不同平台的应用服务可以互操作。利用 Web 服务，可以实现跨操作系统、跨应用程序、跨编程语言的应用程序互操作。例如，一个运行在 Linux 操作系统上的 Java 程序可以调用一个运行在 Windows 操作系统上的用.NET 开发的 Web 服务，反之亦然。

1.4.1　Web 服务简介

Web 服务实现了在异类系统之间以 XML 消息的形式进行数据交换。在 Web 服务出现以前，人们也曾尝试过几种在不同应用程序之间通信的方式，如 DCOM、IIOP 和 Java/RMI 等，这些方式要求在客户端和服务器之间进行紧密集成，并使用特定的二进制数据格式。相对于以前出现的这些技术来说，Web 服务对通信双方的要求更加宽松，只要求接收方可以理解收到的消息。

Web 服务的基础是 HTTP 和 XML。Web 服务使用 HTTP 协议进行网络通信，用 XML 描述通信过程中传输的数据。除 HTTP 和 XML 这两种最基础也最常见的技术外，Web 服务还要用到以下 3 个协议：

- ❑ 简单对象访问协议 SOAP（Simple Object Access Protocol）：是一种标准化的通信规范，主要用于 Web 服务（web service）中。SOAP 的出现是为了使 Web 服务器在检索 XML 数据时无须花时间去格式化页面，并能够让不同应用程序之间通过 HTTP 通信协定，以 XML 格式互相交换彼此的数据，使其与编程语言、平台和硬件无关。
- ❑ Web 服务描述语言 WSDL（Web Services Description Language）：WSDL 描述 Web 服务的公共接口。这是一个基于 XML 的关于如何与 Web 服务通信和使用的服务描述；也就是描述与目录中列出的 Web 服务进行交互时需要绑定的协议和信息格式。
- ❑ 统一描述、发现和集成 UDDI（Universal Description, Discovery, and Integration）：是一个基于 XML 的跨平台的描述规范，可以使世界范围内的企业在互联网上发布自己所提供的服务。

1.4.2　创建 Web 服务

由 1.4.1 节讲解的知识来看，Web 服务涉及多个协议的解析处理，是一个很复杂的工作。ASP.NET 对这些协议进行了封装，使开发人员可以将注意力集中于服务本身而非通信协议。当使用 ASP.NET 进行基本的 Web 服务开发时，并不要求开发人员理解这些底层协议，如 SOAP、WSDL 等，开发人员可以方便快速地开发基于 Web 服务的应用程序。本节和 14.1.3 节将通过具体例子来说明如何创建和访问 Web 服务。

【例 1-16】 创建 Web Service。

本例演示如何创建一个 Web Service。本例创建的 Web Service 只有一个功能：将公历日期转换为中国农历日期。

（1）在 Visual Studio 开发环境中，从菜单栏中选择“文件”|“新建”|“项目”命令，在弹出的“新建项目”对话框中，选择“ASP.NET Web 服务应用程序”，单击“确定”按钮。项目创建后，会生动生成一个 Service1.asmx 文件，这就是 Web 服务文件，asmx 是 ASP.NET Web 服务的文件扩展名。

（2）打开 Service1.asmx.cs 文件，可以看到如下代码：

```
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
public class Service1 : System.Web.Services.WebService
{
    [WebMethod]
    public string HelloWorld()
    {
        return "Hello World";
    }
}
```

上述代码中，在 HelloWorld 方法前面的[WebMethod]表示此方法是一个 Web 服务方法，客户端可以通过 Web 服务方式调用此方法。自动生成的 class Service1 前面的代码现在暂时不需要理解。

（3）修改 Service1.asxm.cs 的代码。删除自动生成的 HelloWorld 方法，添加一个 GetLunarDate()方法。注意要在 GetLunarDate 前面添加 WebMethod 标志。代码如下：

```
//得到农历日期的 Web 服务
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
public class Service1 : System.Web.Services.WebService
{
    //汉字数字数组
    string[] numbers =
("零，一，二，三，四，五，六，七，八，九，十，十一，十二，十三，十四，十五"
+"，十六，十七，十八，十九，二十，廿一，廿二，廿三，廿四，廿五，廿六，廿七"
+"，廿八，廿九，三十").Split('，');
    /// Web 服务方法，以字符串形式返回公历日期对应的农历日期
    /// <param name="date">要转换的公历日期</param>
    /// <returns>转换后的农历日期</returns>
    [WebMethod]
    public string GetLunarDate(DateTime date)
    {
        System.Globalization.ChineseLunisolarCalendar calendar = new
        System.Globalization.ChineseLunisolarCalendar();
        string result ="";
        result = result + toChineseNum(calendar.GetYear()) + "年";
                                              //得到农历年
        result = result + numbers[calendar.GetMonth(date)]+"月";
                                              //得到农历月
        int month=calendar.GetDayOfMonth(date);     //得到农历日
        //如果农历日小于 10，则按照习惯，应该在对应汉字前加一个“初”字
        //如果农历日大于 10，则不加“初”字
```

```
        if (month < 11)
            result = result + "初" + numbers[month];
        else
            result = result + numbers[month];
        return result;
    }
    // 将整数转换为汉字数字形式（如 2010 转换为二零一零）
    private string toChineseNum(int n)
    {
        int r;
        string result = "";
        while (n > 0)                          //此循环对每位数字进行转换
        {
            r = n % 10;
            result = numbers[r] + result;
            n /= 10;
        }
        return result;
    }
}
```

（4）在 Service1.asmx 文件上右击，从弹出的快捷菜单中选择“在浏览器中查看”选项，则可以在浏览器中打开这个 Web 服务，页面中显示了对于此服务的简单说明，如图 1.19 所示。



图 1.19　Service1 Web 服务页面

从图 1.19 所示的 Service1.asmx 页面中可以看到，Service1 包含一个 GetLunarDate()方法，这就是在代码中标识为 WebMethod()的方法。

（5）在图 1.19 所示的 Service1.asmx 页面中，单击页面顶部的“服务说明”超链接，可以转到此服务对应的 WSDL 页面，如图 1.20 所示。

从图 1.20 可以看到，ASP.NET 自动生成了 Web Service 的 WSDL，从而把开发人员从底层协议中解放出来。

（6）在图 1.19 所示的 Service1.asmx 页面中，单击 GetLunarDate()方法链接，会转到此

方法的说明页面，如图 1.21 所示。

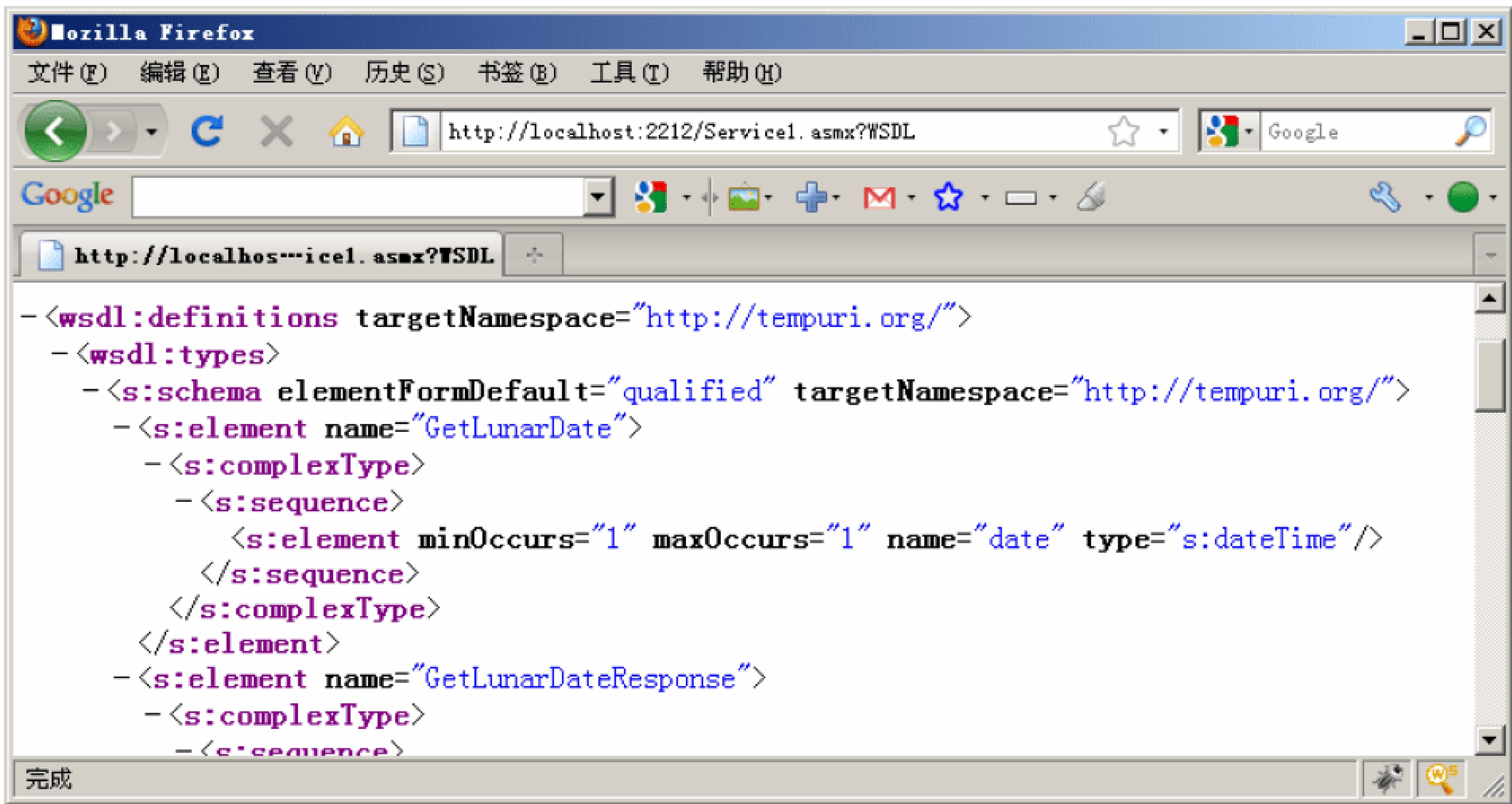


图 1.20　Service1 的 WSDL 页面

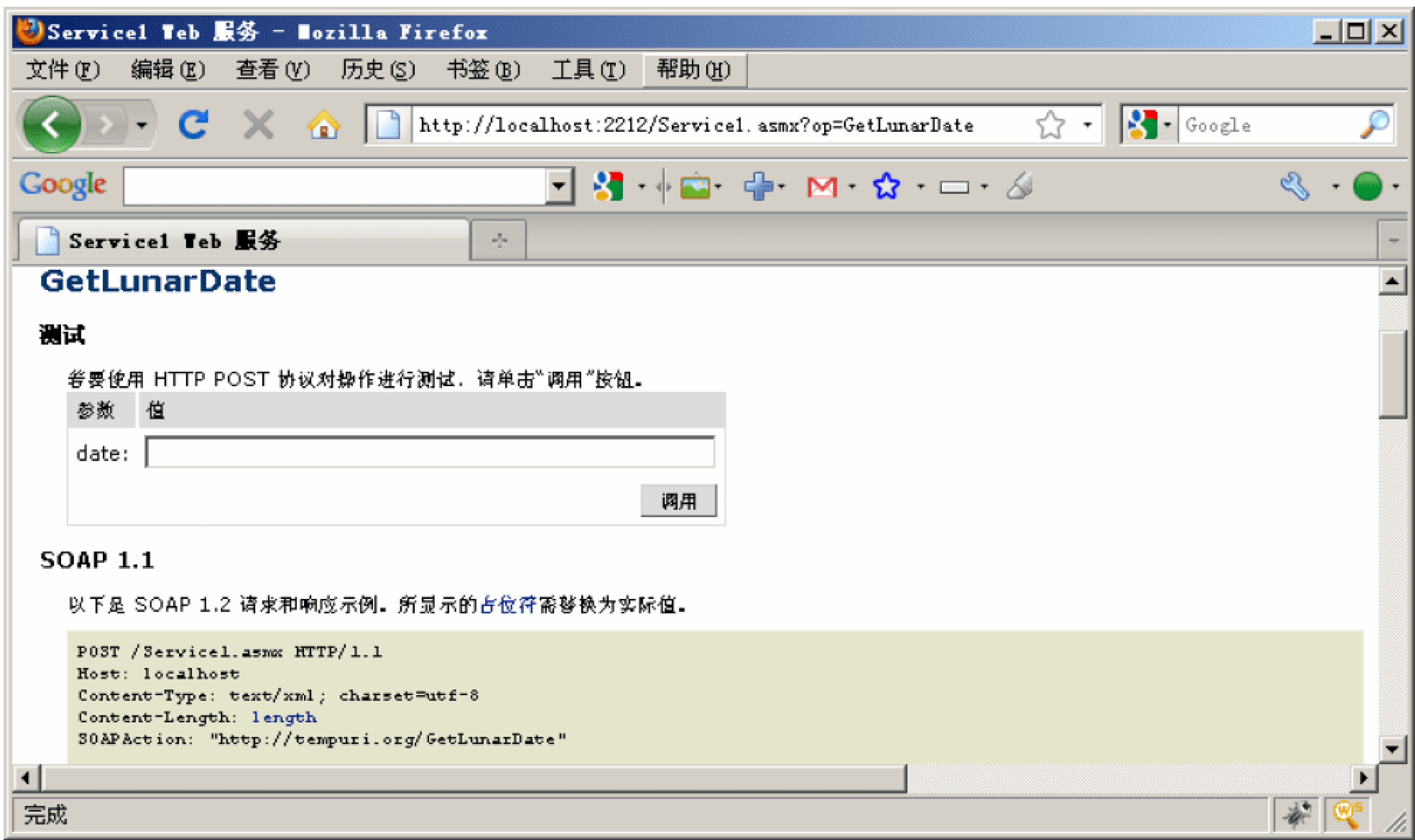


图 1.21　GetLunarDate()方法说明页面

在图 1.21 所示页面中，显示了 GetLunarDate()方法所需的参数，还显示了此方法对应的 SOAP 消息示例。在此页面上，给 GetLunarDate()方法输入参数，单击“调用”按钮，即可通过 Web 服务调用此方法，并在浏览器中显示返回结果，如图 1.22 所示。

（7）如果要让 Web Service 脱离开发环境运行，则需要将其发布。发布 Web Service 的方法与发布 Web 应用程序相同。在 Visual Studio 解决方案资源管理器中，右击 Web Service 项目名称（本例为 WebService1），从弹出的快捷菜单中选择“发布”选项，则弹出如图 1.23 所示的“发布 Web”对话框。从“发布方法”下拉列表框中选择“文件系统”，在“目标位置”文本框中设置发布路径后，单击“发布”按钮，即可完成发布。

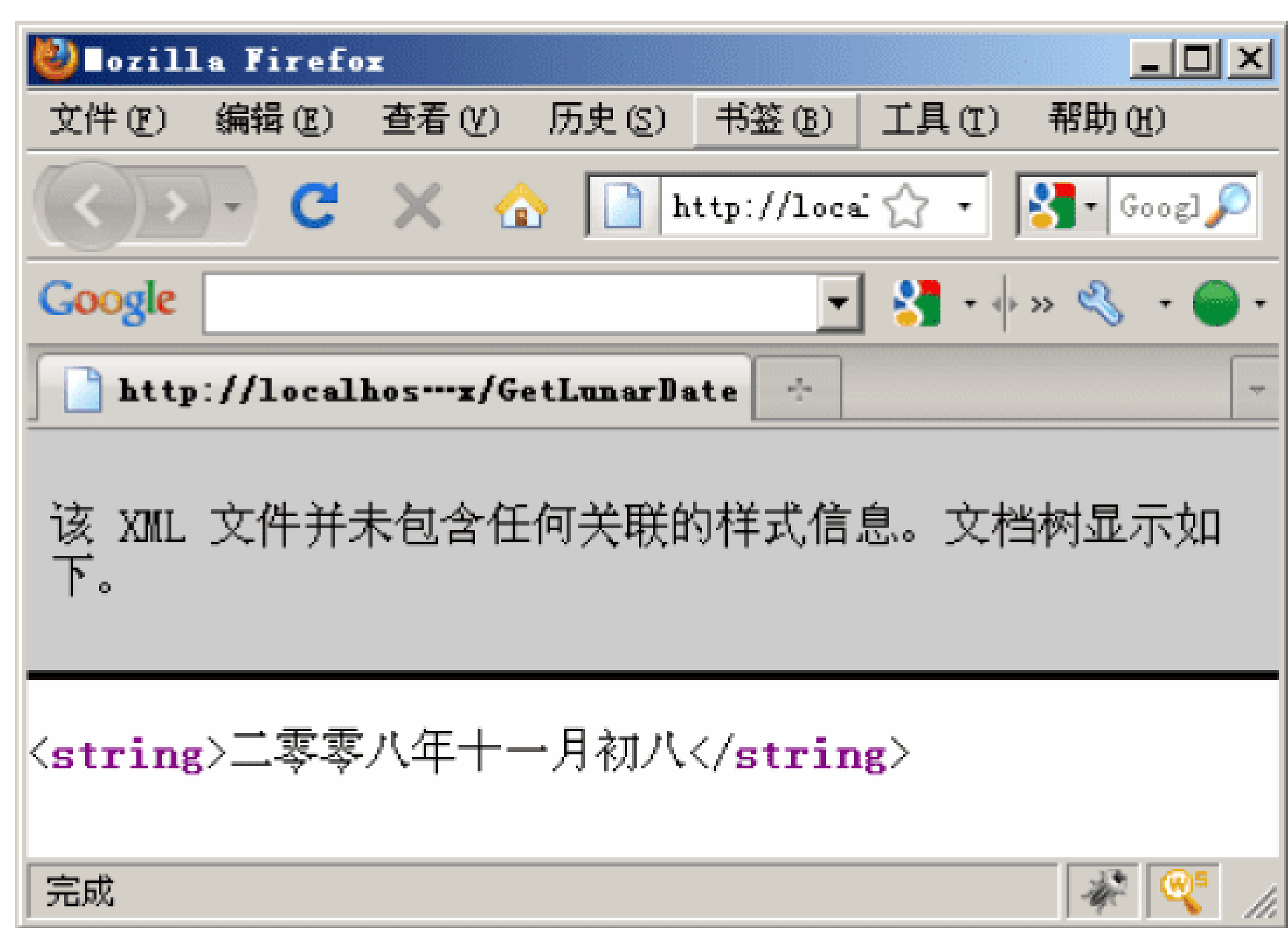


图 1.22　GetLunarDate()方法调用结果

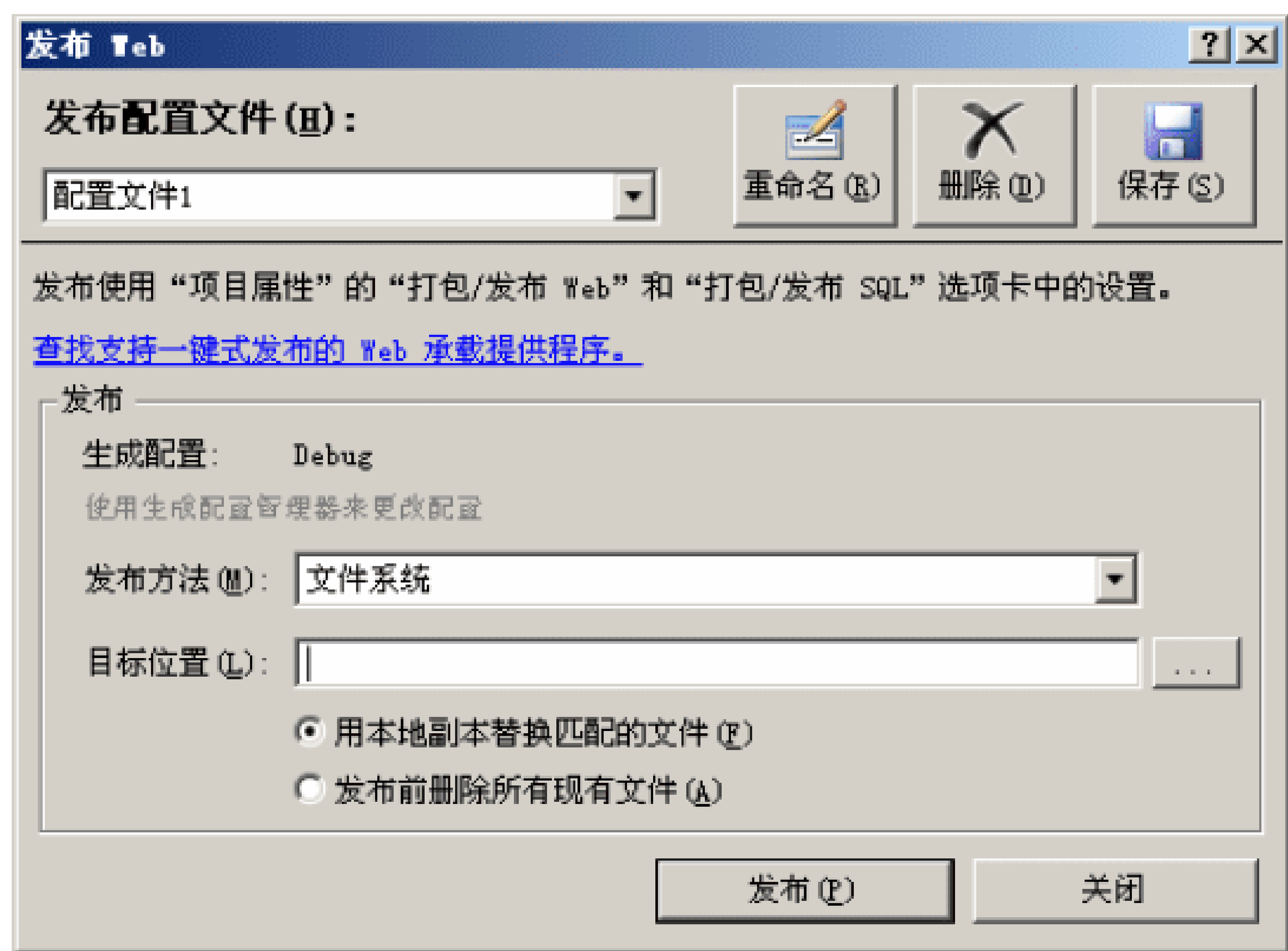


图 1.23　发布 Web 服务

（8）在 Visual Studio 中完成 Web Service 的发布后，会把 Web Service 相关文件都复制到用户所指定的目录下。接下来，还需要将其部署到目标服务器上。主要操作步骤是，找到发布的目标文件夹，把文件夹复制到 Web 服务器上，并将此文件夹设置为 Web 共享，设置相应的访问权限（如是否允许匿名访问、列出目录、读写权限等）。

提示：如何判断一个 ASP.NET Web 服务是否部署成功？方法是在浏览器中输入 Web 服务地址，如果出现了 Web 服务描述页面（如图 1.19 所示），则 Web 服务部署成功。

1.4.3　访问 Web 服务

1.4.2 节介绍了创建 Web 服务的过程。Web 服务创建后，公开了一系列操作接口，各种客户端应用程序（包含 Web 应用、PC 上的 WinForm 应用、移动设备应用程序）都可以按照 Web 服务标准访问此接口，享受 Web 服务所提供的功能。

如前所述，由于 ASP.NET 对 Web 服务底层协议进行了很好的封装，开发人员开发 Web 服务应用程序变得简单高效。从 ASP.NET 中访问一个 Web 服务也很容易实现。下面通过一个例子说明如何访问 Web 服务。

【例 1-17】　访问 Web 服务。

本例演示如何访问例 1-16 所创建的公历日期转农历日期的 Web 服务。

（1）创建一个 ASP.NET Web 应用程序，命名为 WebServiceClient。

（2）运行例 1-16 所创建的 Web 服务。运行这个服务有两种方式，一种是从 Visual Studio 开发环境中运行，一种是将其发布和部署到 Web 服务器再运行。这两种方式都可以，但是一定要保证 Web 服务正确运行。检测 Web 服务是否正常运行的方法是在浏览器中查看此服务，如果服务正常运行了，把服务的 URL 地址复制下来，在下一步骤中将用到此地址。

（3）在 WebServiceClient 项目中右击，在弹出快捷菜单中选择“添加 Web 引用”命令，则弹出“添加 Web 引用”对话框。在对话框的地址栏中输入前一步所获得的 Web 服务地址，则对话框中会显示与此 Web 服务相关的描述信息，如图 1.24 所示。

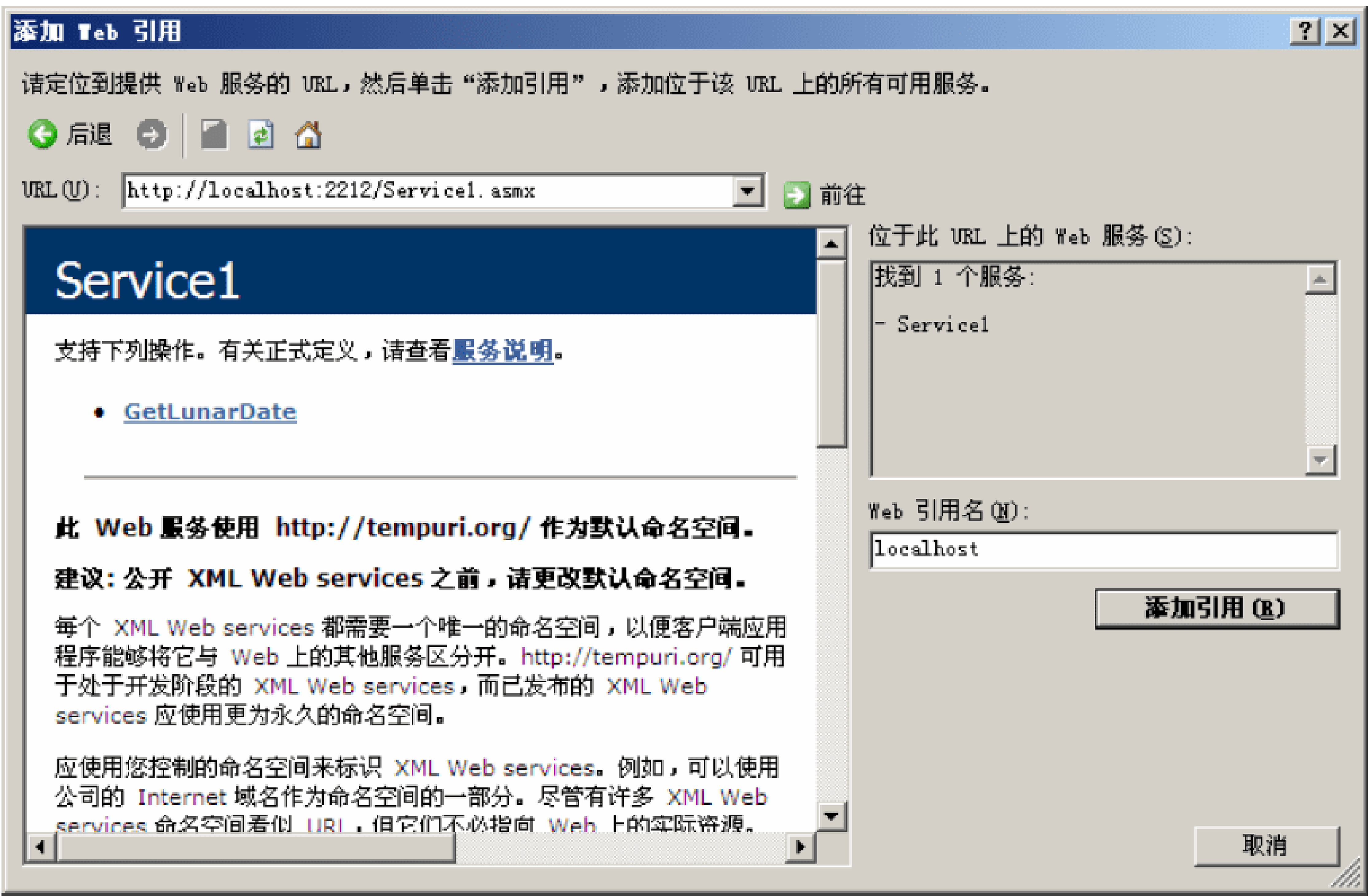


图 1.24　添加 Web 引用

（4）在图 1.24 所示对话框中单击“添加引用”按钮，则 Visual Studio 会自动生成一组类，这些类封装了调用指定的 Web 服务所需要的信息。可以通过对象浏览器窗口查看自动生成的类的相关信息，如图 1.25 所示。

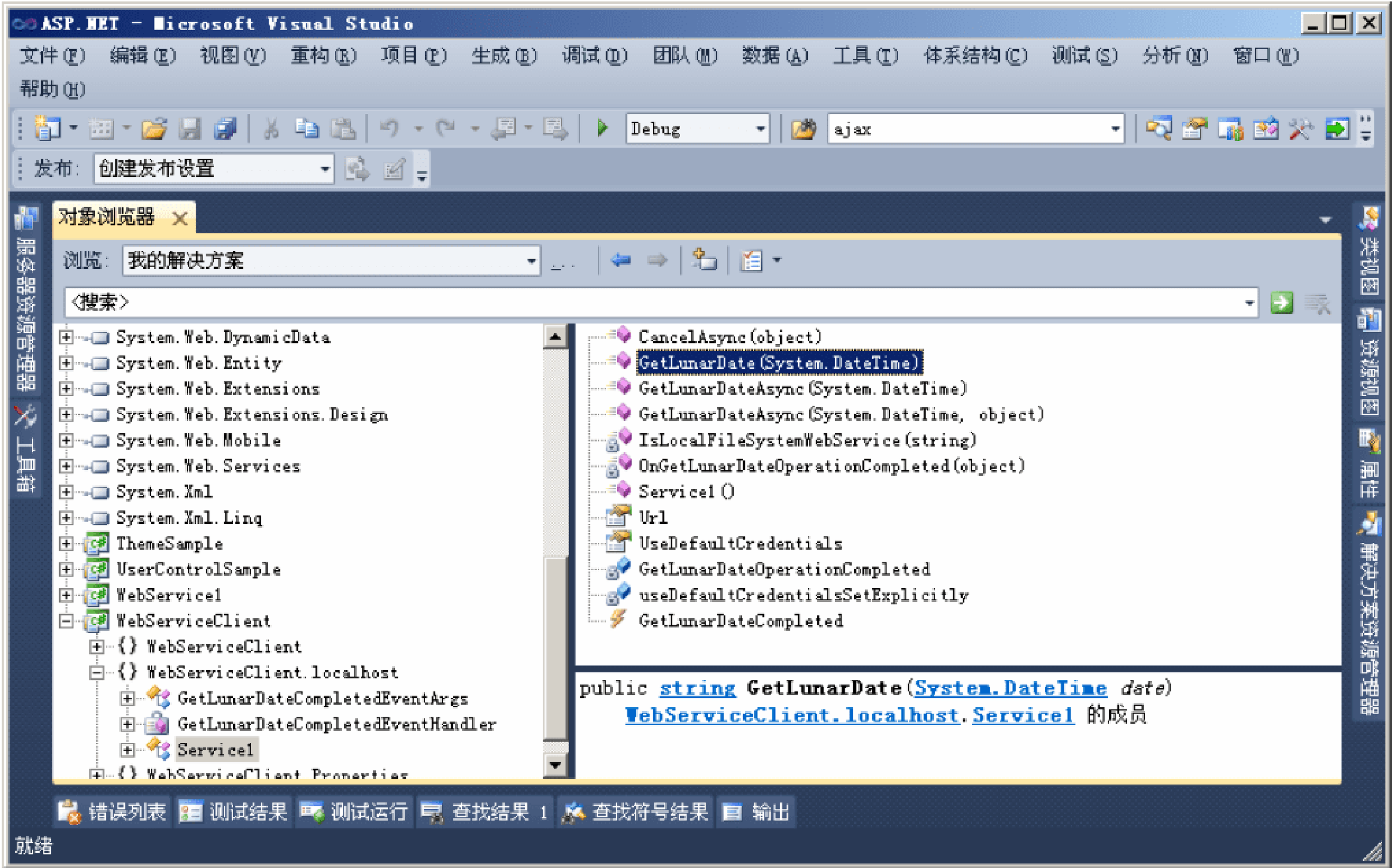


图 1.25　添加 Web 引用后自动生成的类

图 1.25 中，自动生成的这些类的名字与创建 Web Service 时使用的类名相同，但却是不同的类。在客户端自动生成的类其实是服务器端相应类的代理，而不是服务器端类本身。客户端代理类模拟了服务器端相应类的接口，从而允许用户像使用服务器端类那样来使用代理类。另外，代理类在 Web Service 中起着通信桥梁的作用，客户通过代理类向 Web 服

务发出请求，Web 通过代理向客户返回结果。

🔔**注意：** 添加 Web 引用后自动生成的类与 Web 服务中的类虽然名字相同，但却是不同的类，前者是后者的代理，要注意区分这两者。

（5）在 WebServiceClient 项目中添加一个页面 Default.aspx，页面上放置一个 Calendar 和一个 Label，页面代码如下：

```
<form id="form1" runat="server">
<div>选择一个阳历日期查看其对应的农历日期:
    <asp:Calendar ID="Calendar1" runat="server"
        onselectionchanged="Calendar1_SelectionChanged"></asp:Calendar>
        <br />
    <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
</div>
</form>
```

（6）当用户从 Calendar 控件中选择一个日期时，后台程序会调用 Web 服务把此日期转换为农历并显示在页面上。为此，需要在 Calendar 控件的 SelectionChanged 事件中编写如下代码。

```
protected void Calendar1_SelectionChanged(object sender, EventArgs e)
{
    //创建一个 Web Service 代理类的实例
    localhost.Service1 service = new WebServiceClient.localhost.
    Service1();
    //调用 Web Service，得到农历日期
    string lunar=service.GetLunarDate(Calendar1.SelectedDate);
    //把阳历和农历日期都显示在 Label 上
    Label1.Text = string.Format("{0:D}所对应的农历是{1}", Calendar1.Selected
    Date, lunar);
    service.Dispose();                                    //释放代理资源
}
```

（7）运行此页面，从 Calendar 控件中选择一个日期，下方就会显示出对应的农历日期，如图 1.26 所示。

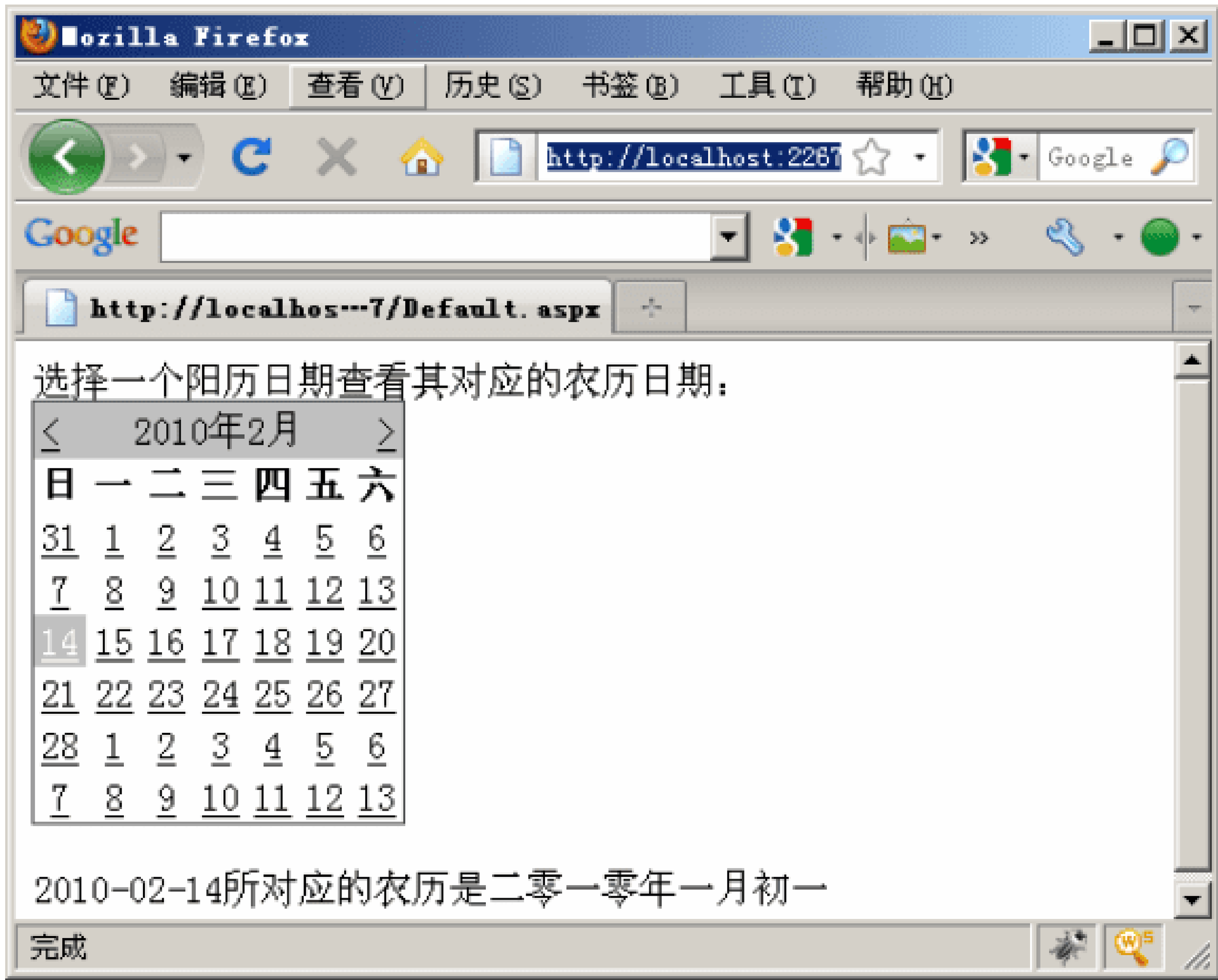


图 1.26　Web 服务客户端运行界面

1.4.4　Web Service 实例——生活小助手

软件的集成和互操作是软件发展的一个重要方向，Web 服务正是顺应了这一方向，提供了跨平台的软件互操作手段。Web 服务的应用发展迅速，当前的 Internet 有许多公司提供各种 Web 服务，有的服务收费，也有免费服务，这些服务内容包含中英文互译、天气预报、股市行情查询、邮政编码查询、电话区号查询、飞机航班、列车时刻查询、IP 地址查询等。基于 Internet 存在的众多 Web 服务，程序员可以轻松开发出功能强大的应用程序。

【例 1-18】 生活小助手。

本例利用当前 Internet 上存在的一些免费 Web 服务，开发了一个小实用程序，该程序可以实现中英文互译和天气预报功能。这两个功能都是人们在工作和生活中经常用到的，所以把这个程序称为生活小助手。

由于本例依赖于 Internet 上现存的 Web 服务，如果被依赖的 Web 服务发生更改或者关闭，那么本例将不能正常运行。如果读者在阅读本书时，本例所引用的 Web 服务已经不能使用，读者可以从网上搜索其他类似的 Web 服务，按照本例所讲的方法，参照例子中的代码，实现类似的功能。

（1）创建一个 ASP.NET Web 应用程序，命名为 Assistant。

（2）生活小助手程序包含两个功能，首先实现英汉互译功能。在项目中添加英汉互译的 Web 服务引用，地址为 http://fy.webxml.com.cn/webservices/EnglishChinese.asmx。这个 Web 服务包含多个方法，本例主要用到其中的一个 TranslatorString()方法，此方法接收一个 string 类型参数，表示需要翻译的词语（中英文都可以），返回一个字符串数组，表示翻译结果。

（3）在项目中添加一个 Default.aspx 页面。

（4）这个程序包含两个功能，每个功能在页面上表现为一个 div，为了实现统一布局，在页面中定义一个 CSS，代码如下：

```
div.assistant
{
   border:dashed 1px silver;                          /*边框银灰色细虚线*/
   width:300px;height:150px;                          /*宽 300，高 150*/
   background-color:#ccddff;                          /*背景浅蓝色*/
   margin:10px;                                       /*边距为 10 个像素*/
   float:left;                                        /*向左浮动*/
}
```

（5）在 Default.aspx 页面中添加一个 div 以及相关控件，实现翻译界面。代码如下：

```
<div class="assistant">
<h3>英汉双向翻译</h3>
输入单词：<asp:TextBox runat="server" ID="word" />
   <asp:Button ID="Button1" runat="server" Text="翻译" onclick="Button1
   Click" /><br />
   <asp:Label ID="translation" runat="server" ></asp:Label>
</div>
```

（6）为“翻译”按钮编写代码，调用 Web Service 实现翻译功能并显示结果，代码

如下：

```
protected void Button1 Click(object sender, EventArgs e)
{
    webservice.EnglishChinese utility = new
         BusinessAssistant.webservice.EnglishChinese();
                                                    //生成 Web 服务代理
    try
    {
        string[] result = utility.TranslatorString(word.Text);
                                                    //调用 Web 服务进行翻译
        //把翻译结果显示在页面上
        string temp = "";
        temp += "读音： " + result[1] + "<br/>";
        temp += "翻译： " + result[3] + "<br/>";
        translation.Text = temp;
    }
    catch
    {
        translation.Text = "翻译过程中出现错误。"; //如果出现异常，则显示提示信息
    }
    finally
    {
        utility.Dispose();
    }
}
```

（7）运行 Default.aspx 页面，运行结果如图 1.27 所示。

（8）前面几个步骤实现了英汉互译功能，下面再来实现天气预报功能。本例所选用的天气预报 Web 服务地址为 http://www.ayandy.com/Service.asmx，在项目中添加对这个 Web 服务的引用。这个 Web 服务包含多个方法，本例只用到其中一个 getWeatherByCityName() 方法。此方法接收两个参数：第一个参数为 string 类型，表示要查询的城市名称，第二个参数为枚举类型，表示要查询哪一天的天气（可以是今天、明天或者后天）。此方法返回一个字符串数组，表示查询到的结果。

（9）在 Default.aspx 页面添加另一个 div，并在其中放置一个 TextBox、一个 Label 和一个 DropDownList，作为天气预报查询界面。代码如下：

```
<div class="assistant">
<h3>天气预报查询</h3>
要查询的城市：<asp:TextBox runat="server" ID="city" /><br />
要查询的日期：<asp:DropDownList ID="date"
    runat="server">
          <asp:ListItem Value="0">今天</asp:ListItem>
          <asp:ListItem Value="1">明天</asp:ListItem>
          <asp:ListItem Value="2">后天</asp:ListItem>
</asp:DropDownList>
    <asp:Button ID="Button2" runat="server"
    Text="查询" onclick="Button2_Click"  /><br />
    <asp:Label ID="weather" runat="server" ></asp:Label>
</div>
```

（10）为“查询”按钮编写代码，实现天气预报查询和显示，代码如下：

```
protected void Button2_Click(object sender, EventArgs e)
{
```

```
    //生成 Web 服务代理
    webservice1.Service service = new BusinessAssistant.webservice1.
    Service();
    int day = int.Parse(date.SelectedValue);        //得到用户所选择的日期
    try
    {
        //调用 Web 服务获得天气预报
        string[] result = service.getWeatherbyCityName(city.Text,
            (webservice1.theDayFlagEnum)day);
        string temp;
        //得到城市名称和日期
        temp = string.Format("<br/>{0}{1}的天气情况<br/>",result[1],result
        [5]);
        temp += result[2] +";"+ result[3]+";"+result[4]+";<br/>";
                                                    //得到天气、温度和风力
        temp += "<img src='" + result[6] + "' alt='天气图片'/>";
                                                    //得到天气图片
        weather.Text = temp;                        //把以上信息显示在页面上
    }
    catch                                           //异常处理
    {
        weather.Text = "查询过程中出现错误。";
    }
    finally
    {
        service.Dispose();                          //释放资源
    }
}
```

（11）运行 Default.aspx 页面，运行结果如图 1.28 所示。

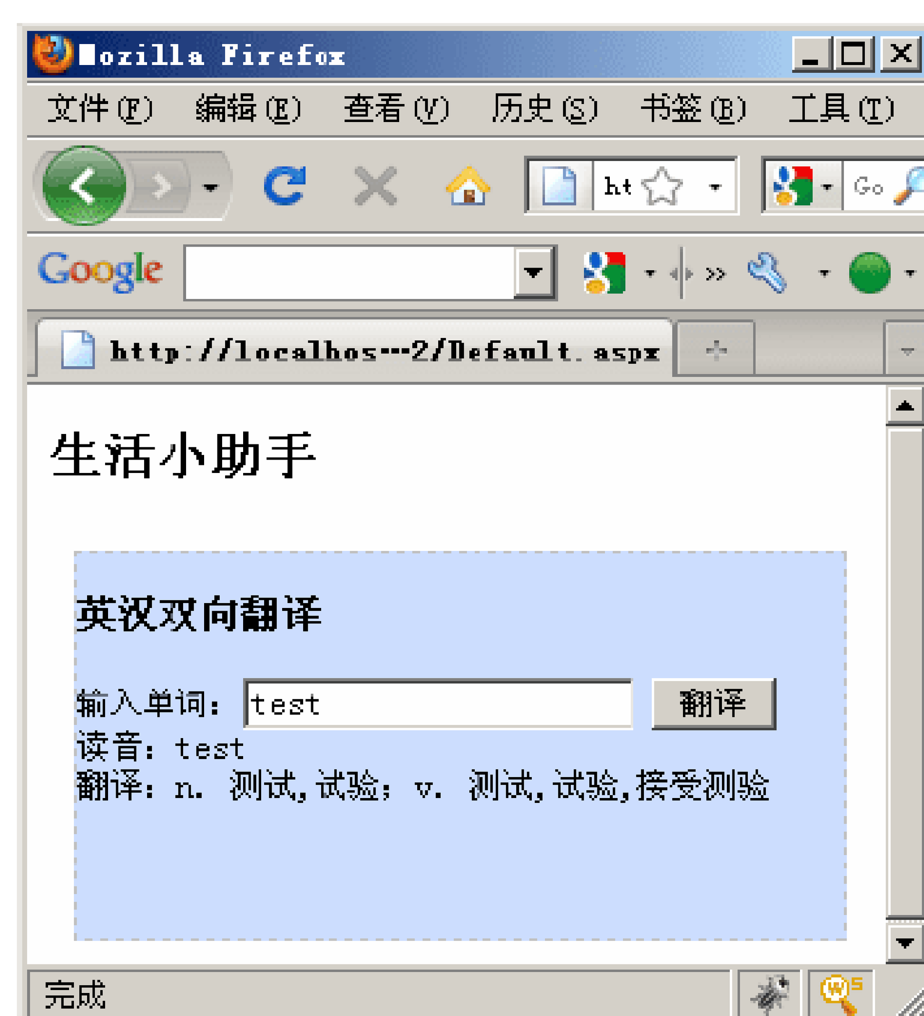


图 1.27　翻译助手运行界面

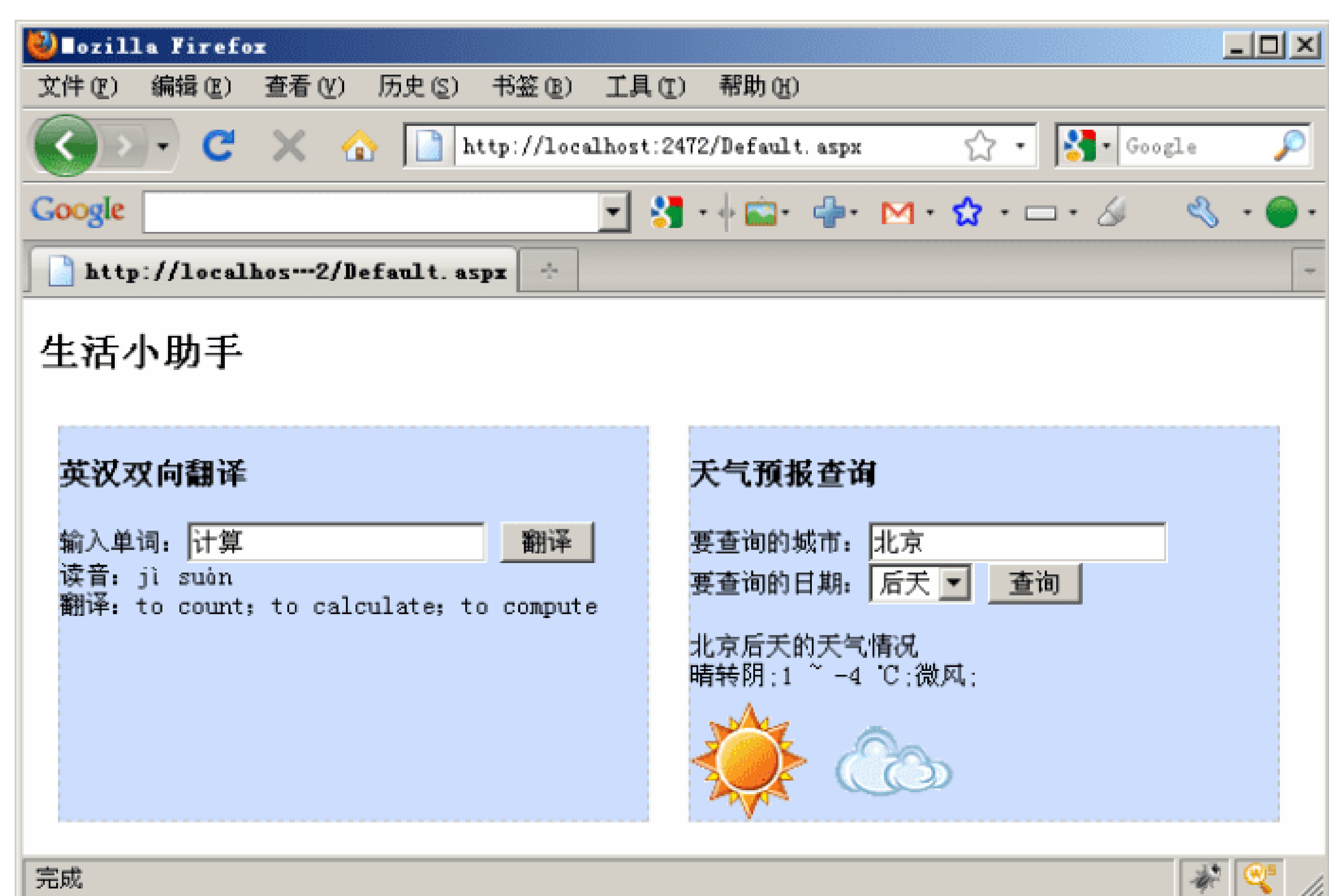


图 1.28　生活小助手运行界面

1.5　用 户 控 件

ASP.NET 用户控件与完整的 ASP.NET 页面相似，可以同时具有用户界面页和代码。用户控件就像是一个局部的 ASP.NET 页面，包含其他子控件，这些子控件也有事件和事件

处理程序。使用用户控件可以提高程序的模块化和复用性。与页面所不同的是，用户控件不是一个完整页面，不能在浏览器中进行浏览。用户控件文件的扩展名是 ascx，ascx 文件的第一条语句是<%@Control 指令。

1.5.1　创建和使用用户控件

相对于 ASP.NET 自带的系统控件来说，用户控件是开发人员自行定义和创建的控件。用户控件继承自 System.Web.UI.UserControl 类。创建用户控件的过程与创建普通 ASP.NET 页面很相似。用户控件不能作为单独页面运行，必须嵌入到 ASP.NET 页面中使用。用户控件可以包含其他控件，包含 ASP.NET 标准控件和其他用户控件。使用用户控件时必须先注册再使用。下面通过一个例子演示如何创建和使用用户控件。

【例 1-19】 创建和使用用户控件。

许多 Web 应用程序的页头和页脚都是固定的内容，页头一般是网站横幅和链接，页脚一般是版权信息和联系信息。可以把这两部分内容分别做成用户控件，提高页面模块化程度和可读性。本例演示如何创建和使用页头、页脚用户控件。本例来源于笔者开发的《新型农村合作医疗管理信息系统》项目，做了少许改动。由于本例代码较多，首先给出例子完成后的界面如图 1.29 所示，使读者有一个整体概念，然后再逐步介绍其实现过程。



图 1.29　页头页脚用户控件示例

（1）创建一个 Web 应用程序，命名为 UserControlSample。

（2）在“解决方案资源管理器”中，右击项目名称，从弹出的快捷菜单中选择“添加”|“新建项”命令，在弹出窗口中选择“Web 用户控件”，重命名为 Weather，然后单击“添加”按钮，如图 1.30 所示。

（3）Weather 用户控件的作用是通过 Web 服务获取和显示天气信息。控件采用 Table 布局，把天气文字信息和图片信息整齐地显示在页面上。通过 Web Service 获取天气的具体实现原理在 Web Service 一节中已经介绍，这里仅给出主要代码。Weather.ascx 的关键代码如下：

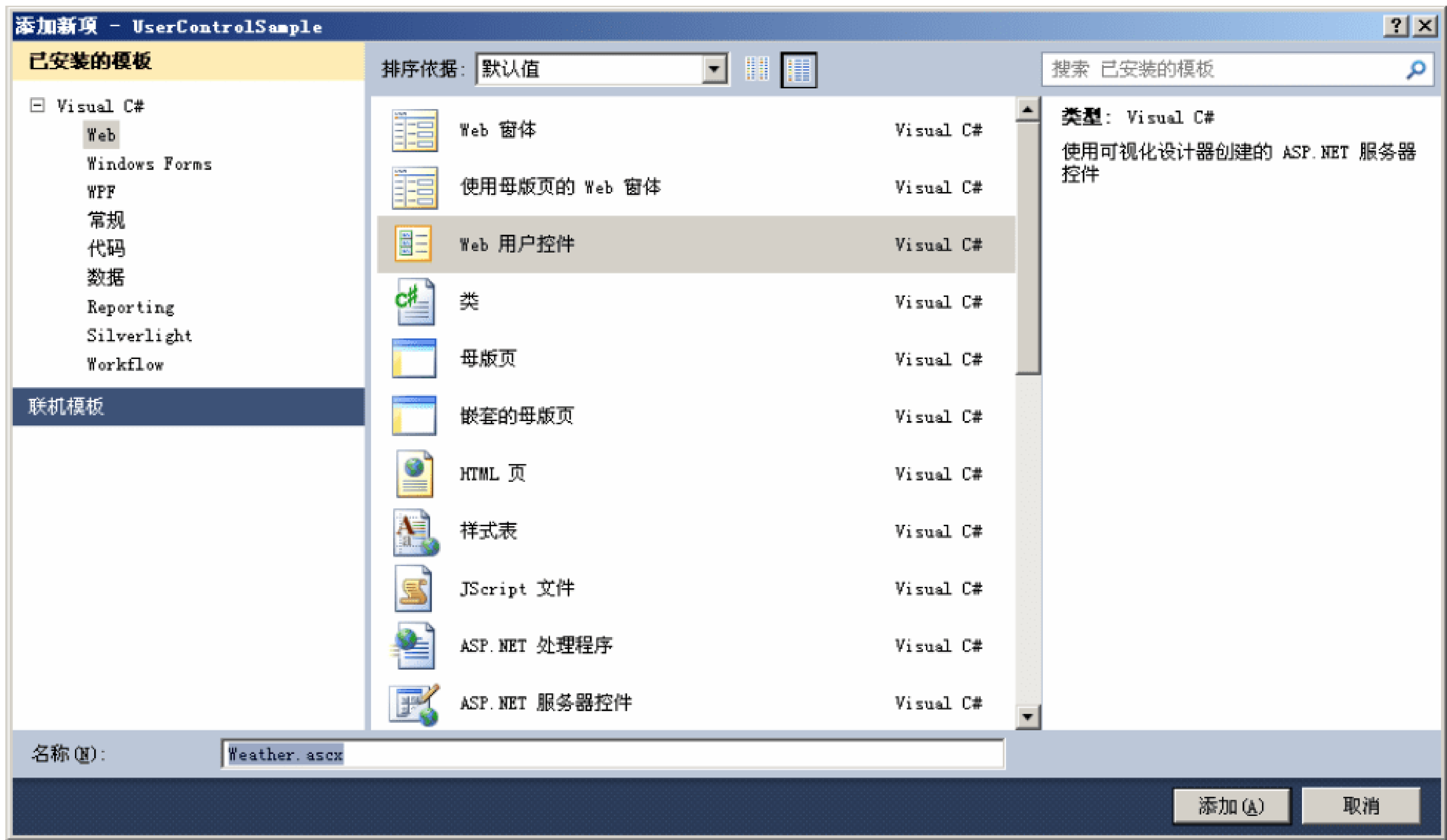


图 1.30　添加 Web 用户控件

```
<table>
<tr>
<td><asp:Label ID="cityName" runat="server" Text="城市名称"></asp:Label><br
/>天气</td>
<td><asp:Image  ID="weatherImage"  runat="server"  Height="32px"  Alternate
Text="天气"  /></td>
<td>
<asp:Label  ID="weatherDesc"  runat="server"  Text="天气情况"></asp:Label>
</td>
</tr>
</table>
```

Weather 用户控件后台代码文件 Weather.ascx.cs 代码如下：

```
protected void Page_Load(object sender, EventArgs e)
{
    showWeather();
}
//在页面上显示天气信息
private void showWeather()
{
    try
    {
        var service = new weather.Service();             //创建 Web 服务代理
        //得到某城市（此处为北京）当天天气信息
        string [] result=service.getWeatherbyCityName("北京", weather.
        theDayFlagEnum.Today);
        service.Dispose();                               //释放资源
        //在页面上显示天气描述信息和图片信息
        weatherDesc.Text=result[2]+"<br/>"+result[3];
        weatherImage.ImageUrl=result[6];
    }
    Catch                                                //发生异常
    {
        weatherDesc.Text = "未能获取天气信息";
```

```
    }
}
```

（4）在项目中添加一个新的用户控件 Header.ascx，用于显示页面头部信息。Header 用户控件包含了 Weather 用户控件，下面介绍如何把 Weather 用户控件添加到 Header 用户控件中。打开 Header.ascx 页面并切换到设计视图，然后从“解决方案资源管理器”中，把 Weather.ascx 用户控件拖动到 Header.ascx 的设计视图，从而完成把 Weather 添加到 Header。此时打开 Header.ascx 文件，可以发现有以下三行代码。

```
<%@  Control  Language="C#"  AutoEventWireup="true"  CodeBehind="Header.
ascx.cs"
 Inherits="UserControlSample.UserControls.Header" %>
<%@ Register src="Weather.ascx" tagname="Weather" tagprefix="uc1" %>
<uc1:Weather ID="Weather1" runat="server" />
```

其中第一行代码是<%@Control 指令，说明这是一个用户控件，还说明了控件的一些基本信息。第二行语句是 Weather 用户控件的注册指令，在当前页面（控件）中注册了 Weather 用户控件。第三条语句添加了一个 Weather 控件。

（5）修改 Header.ascx 控件源码，使其包含主要功能的链接，代码如下：

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="Header.ascx.
cs" Inherits="MyWeb.UserControls.Header" %>
<%--注册天气预报用户控件--%>
<%@ Register src="Weather.ascx" tagname="Weather" tagprefix="uc1" %>
<div style="background:#f2f2fe; width:1000px; text-align:left;">
    <%--最外层 div--%>
<div style="float:left">                                   <%--LOGO，向左浮动--%>
<img src="../images/title.png" alt="" /> <span style="margin-left:180px;
margin-right:30px;">
当前日期：
<%--调用 C#代码显示当前日期和时间--%>
<%=DateTime.Now.ToLongDateString()+" "+DateTime.Today.DayOfWeek %>
</span>
</div>
   <uc1:Weather ID="Weather1" runat="server" />   <%--天气预报控件--%>
</div>
<%--以下为页面顶部的导航条，每个导航项目包括一个图片和一个文字--%>
<div id="NavMenu">
<div class="ImageMenu"><img src="../images/user.png" alt="农民档案"/><br/>
<a href="../People/FamilyPage.aspx">农民参合档案</a></div>
<div  class="ImageMenu"><img  src="../images/medicine.png"  alt="药品目录"
/><br/>
<a href="../Medicine/MedicinePage.aspx">药品检查目录</a></div>
<div  class="ImageMenu"><img  src="../images/hospital.png"  alt="医疗机构"
/><br/>
<a href="../Hospital/HospitalPage.aspx">定点医疗机构</a></div>
<div  class="ImageMenu"><img  src="../images/policy.png"  alt="报销政策"
/><br/>
<a href="../Policy/CopensateRatePage.aspx">报销政策管理</a></div>
<div class="ImageMenu"><img src="../images/dictionary.png" alt="字典维护"
/><br/>
<a href="../Dictionary/DictionaryPage.aspx">数据字典维护</a></div>
```

```
<div class="ImageMenu"><img src="../images/lock.png" alt="用户权限" /><br/>
<a href="../UserRight/ChangePassPage.aspx">用户权限管理</a></div>
</div>
```

（6）在项目中添加页脚控件 Footer.ascx，页脚控件主要包含一些说明信息，代码如下：

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="Footer.ascx.
cs" Inherits="MyWeb.UserControls.Footer" %>
<div style="width:1000px; height:30px; font-size:11px; color:#666; text-
align:center;
   border:dashed 1px silver; background:#f2f2f2; float:none; clear:both;">
版权所有：XXXXX 公司。地址：XX 省 XX 市 XXXXXX 街道 XX 号。联系电话：010-12345678
<br />
技术支持：<a href="mailto:youremail@gmail.com">给我发邮件</a></div>
```

（7）在项目中添加一个页面 XNH.aspx，打开此页面并切换到设计视图，从“解决方案资源管理器”中把 Header 和 Footer 控件拖动到页面中。页面关键代码如下：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="xnh.aspx.cs"
Inherits="UserControlSample.xnh" StylesheetTheme="default" %>
<!--注册 Header 控件-->
<%@ Register src="UserControls/Header.ascx" tagname="Header" tagprefix=
"uc1" %>
<!--注册 Footer 控件-->
<%@ Register src="UserControls/Footer.ascx" tagname="Footer" tagprefix=
"uc2" %>
...
<body>
   <form id="form1" runat="server">
   <div>
   <uc1:Header ID="Header1" runat="server" />      <!--创建 Header 控件-->
   <div style="margin:3px; border:dashed 1px silver; height:100px;">
   <h3>这里是正文内容</h3>
   </div>
   <uc2:Footer ID="Footer1" runat="server" />      <!--创建 Footer 控件-->
   </div>
   </form>
</body>
```

（8）运行 XNH.aspx 页面，运行结果见本例开始处的图 1.29。

1.5.2　添加自定义属性

前面所介绍的 Header 和 Footer 用户控件都不与页面上的其他控件打交道，是比较简单的用户控件。在实际应用中，作为页面的一部分，用户控件和页面上其他控件需要进行交流。ASP.NET 的标准控件如 Button 等，都有一些属性（如 Text）和事件（如 Click），页面代码正是通过这些属性实现与控件的交流和通信。用户控件继承自 System.Web.UI.UserControl，也继承了很多通用属性和事件，但这些属性和事件通常并不能满足要求，这时就需要开发人员自行定义属性和事件。

【例 1-20】 用户控件的属性。

本例将开发一个进度条控件。这个控件有三个自定义属性：当前进度、背景色和前景

色。本例用一个 Table 表示进度条，用 Table 中的单元格 Cell 表示进度块。进度条中共有 20 个进度块，进度条数值范围为 0～100，每 5 个数值对应一个进度块。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加一个 ASP.NET 用户控件，重命名为 ProgressBar，并在其中添加一个服务器端 Table 控件，代码如下：

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="ProgressBar.
ascx.cs" Inherits="UserControlSample.UserControls.ProgressBar" %>
<asp:Table ID="table1" runat="server"  BorderWidth="1px"  CellPadding="1"
CellSpacing="1"  Height="15px"  Width="200px">
</asp:Table>
```

（3）为 ProgressBar 控件添加一个 BackColor 属性，表示进度条背景色。代码如下：

```
//进度条背景色
public Color backColor
{
    get
    {
        //如果 ViewState 中没有保存背景色，则返回默认颜色灰色
        if (ViewState["BackColor"] == null)
            return Color.Silver;
        //如果 ViewState 中已经保存了背景色，则返回保存的颜色
        return (Color)ViewState["BackColor"];
    }
    set
    {
        ViewState["BackColor"] = value;          //把新颜色保存到 ViewState
    }
}
```

注意：由于 Web 应用的无状态性，为控件添加属性时需要考虑属性的持久性，通常可以通过视图状态 ViewState 来实现。

（4）为 ProgressBar 控件添加前景色属性 ForeColor，代码如下：

```
//进度条前景色
public Color foreColor
{
    get
    {
        //如果 ViewState 中没有保存前景色，则返回默认颜色绿色
        if (ViewState["ForeColor"] == null)
            return Color.Green;
        //如果 ViewState 中已经保存了前景色，则返回保存的颜色
        return (Color)ViewState["ForeColor"];
    }
    set
    {
        ViewState["ForeColor"] = value;      //把新颜色保存到 ViewState
    }
}
```

（5）为 ProgressBar 控件添加属性 Progess，表示当前进度。进度条的最大进度为 100，在设置属性时要检测值的合法性。代码如下：

```
//当前进度（有效值为 0 到 100）
public int progress
{
    get
    {
        //如果 ViewState 中没有保存当前进度，则返回 0，否则返回保存的进度
        if (ViewState["Progress"] == null)
            return 0;
        return (int)ViewState["Progress"];
    }
    set
    {
       //如果要设置的值合法（在 0 到 100 之间），则接受，否则拒绝
        if (value > 100 || value < 0) return;
        ViewState["Progress"] = value;
    }
}
```

（6）在 ProgressBar 控件的 PreRender 事件中，根据前景色、背景色、当前进度的值，生成进度条控件界面。代码如下：

```
protected void Page_Load(object sender, EventArgs e)
{    //为控件的 PreRender 事件添加事件处理程序
    this.PreRender += new EventHandler(ProgressBar_PreRender);
}
void ProgressBar_PreRender(object sender, EventArgs e)
{   //获得当前进度所对应的进度块（一块表示 5 个数值）
    int blocks=(int)Math.Round(progress*20.0/100.0);
    TableRow row = new TableRow();                    //创建 Table 中的一行
    //添加单元格，一共添加 20 个
    for (int i = 0; i <20 ; i++)
    {
        TableCell cell = new TableCell();
        cell.Text = " ";
        //根据当前进度设置单元格颜色
        if (i < blocks)
            cell.BackColor = foreColor;
        else
            cell.BackColor = backColor;
        row.Cells.Add(cell);                          //把单元格添加到行
    }
    table1.Rows.Add(row);
}
```

（7）ProgressBar 控件到此为止创建完成，下面编写一个测试页面。在项目中添加一个新的 ASP.NET 页面 ProgressPage.aspx，把 ProgressBar 控件拖到页面上，并在页面上放置按钮等控件以进行测试。ProgressPage.aspx 关键代码如下：

```
<form id="form1" runat="server">
<div>
    <uc1:ProgressBar ID="ProgressBar1" runat="server" />
                                                      <%--进度条用户控件--%>
    <br />
    <asp:Button ID="Button1" runat="server" Text="增加" onclick="Button1_
    Click" />
```

```
    <asp:Button ID="Button2" runat="server" Text="减少" onclick="Button2_
    Click" /><br />
    前景色：
    <asp:DropDownList ID="foreColorList" runat="server">
                                                    <%--前景色下拉列表--%>
          <asp:ListItem Value="Green">绿色</asp:ListItem>
       <asp:ListItem Value="Red">红色</asp:ListItem>
       <asp:ListItem Value="Blue">蓝色</asp:ListItem>
    </asp:DropDownList>
    背景色：
    <asp:DropDownList ID="backColorList" runat="server">
                                                    <%--背景色下拉列表--%>
       <asp:ListItem Value="Black">黑色</asp:ListItem>
       <asp:ListItem Value="White">白色</asp:ListItem>
       <asp:ListItem Value="Silver">灰色</asp:ListItem>
    </asp:DropDownList>
    <asp:Button ID="Button3" runat="server" Text="修改颜色" onclick=
    "Button3_Click" />
</div>
</form>
```

（8）为 ProgressPage.aspx 页面的 3 个按钮编写代码如下：

```
//增加进度，出于演示目的，为当前进度添加一个随机数
protected void Button1_Click(object sender, EventArgs e)
{
    Random r = new Random(DateTime.Now.Second);
    ProgressBar1.progress += r.Next(20);
}
//减少进度，出于演示目的，为当前进度添加一个随机数
protected void Button2_Click(object sender, EventArgs e)
{
    Random r = new Random(DateTime.Now.Second);
    ProgressBar1.progress -= r.Next(20);
}
//改变进度条颜色
protected void Button3_Click(object sender, EventArgs e)
{
    Color fore = Color.FromName(foreColorList.SelectedValue);
    Color back = Color.FromName(backColorList.SelectedValue);
    ProgressBar1.foreColor = fore;
    ProgressBar1.backColor = back;
}
```

（9）运行 ProgressPage.aspx 页面，运行界面如图 1.31 所示。

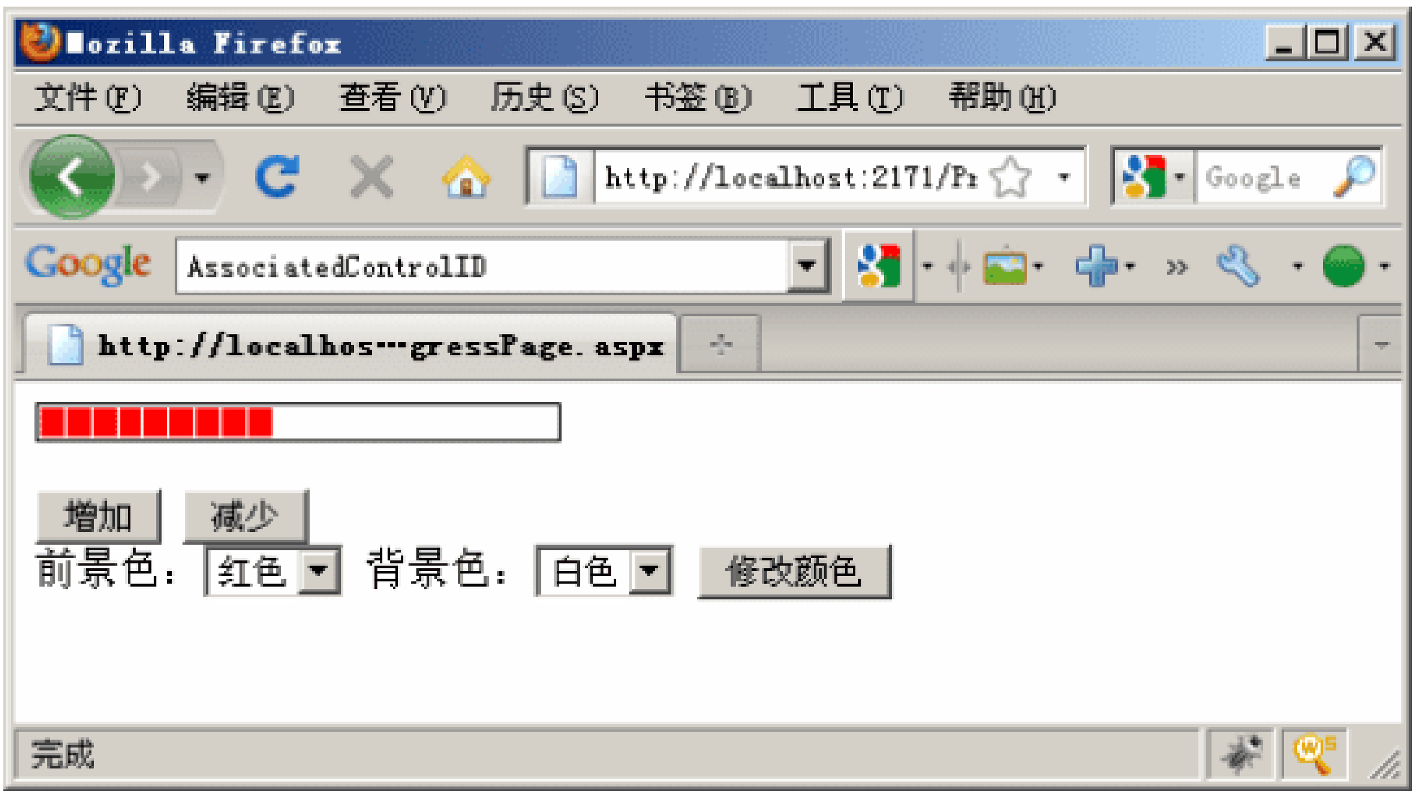


图 1.31　进度条控件示例

1.5.3　添加自定义事件

ASP.NET 标准控件都有各种事件，如 Button 控件有 Click 等事件。控件通过事件机制将自身的状态变化通知外界环境。对于用户控件来说，很多时候也需要事件机制与外界进行通信。

【例 1-21】 用户控件的事件。

本例将制作一个可以复用的登录用户控件，该控件的布局类似于 ASP.NET 标准控件中的 Login 控件，包含两个文本框和一个“登录”按钮。用户输入用户名和密码后，单击“登录”按钮，即可尝试登录。

由于这是一个可以复用的控件，可以被应用于不同的程序中，所以控件的开发者并不知道如何验证用户名和密码，例如，如果用户信息保存在数据库中，那么控件开发者不知道数据库类型、表名、列名等，从而无法对登录信息的合法性进行验证。具体的验证工作应该由控件的使用者而不是开发者来完成。所以，控件的开发者不能在“登录”按钮的 Click 事件中写代码验证登录，而是需要把这个验证交给控件所在的页面来做，这就需要定义一个事件来完成这个功能。

（1）创建一个 ASP.NET 应用程序。

（2）在项目中添加一个用户控件，命名为 MyLogin，并在其中放置用于输入用户名和密码的 TextBox，一个“登录”Button，以及两个验证控件。为了使各个控件排列整齐，使用 table 布局。代码如下：

```
<table border="0" cellpadding="1" cellspacing="0"  style="border-collapse:
collapse;">
<tr><td align="center" colspan="2">登录</td> </tr>
<tr><td align="right">
    <asp:Label  ID="UserNameLabel"  runat="server"  AssociatedControlID=
    "UserName">用户名:</asp:Label></td>
    <td> <asp:TextBox ID="UserName" runat="server"></asp:TextBox>
    <asp:RequiredFieldValidator   ID="UserNameRequired"   runat="server"
    ControlToValidate="UserName" ErrorMessage="必须填写"用户名"。" ToolTip="
    必须填写用户名。" ValidationGroup="Login1">*</asp:RequiredFieldValidator>
    </td>
</tr>
<tr><td align="right">
        <asp:Label ID="PasswordLabel" runat="server" AssociatedControlID=
        "Password">密码:</asp:Label></td>
    <td>
    <asp:TextBox ID="Password" runat="server" TextMode="Password"></asp:
    TextBox>
    <asp:RequiredFieldValidator   ID="PasswordRequired"   runat="server"
    ControlToValidate="Password" ErrorMessage="必须填写"密码"。" ToolTip="
    必须填写密码。" ValidationGroup="Login1">*</asp:RequiredFieldValidator>
</td>
</tr>
<tr> <td align="center" colspan="2" style="color:Red;">
        <asp:Literal ID="FailureText" runat="server" EnableViewState=
        "False"></asp:Literal> </td>
</tr>
<tr><td align="right" colspan="2">
```

```
        <asp:Button ID="LoginButton" runat="server" CommandName="Login"
        Text="登录"
            ValidationGroup="Login1" onclick="LoginButton Click" /></td>
</tr>
</table>
```

控件在设计状态下的外观如图 1.32 所示。

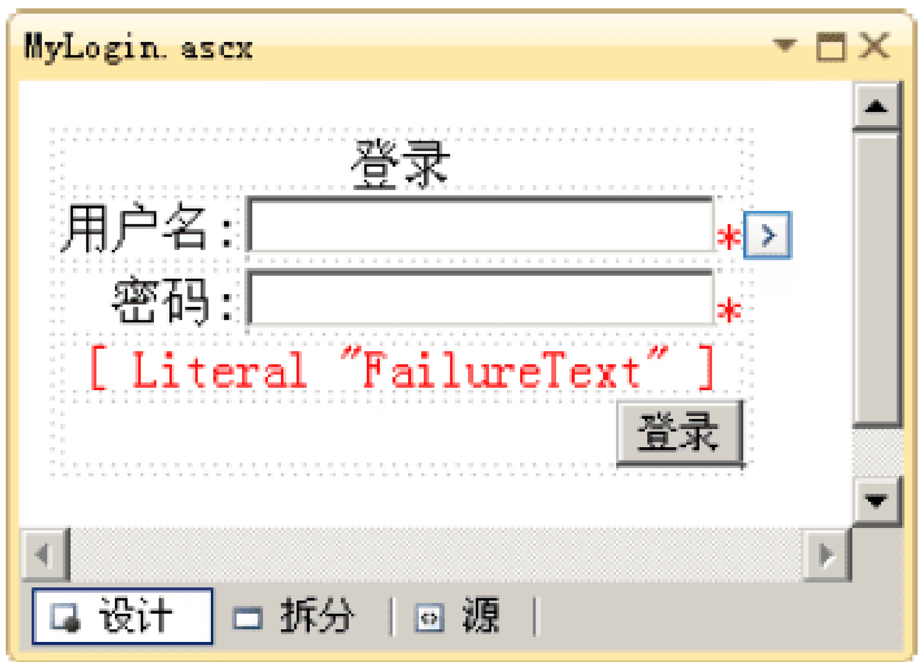


图 1.32　登录控件设计界面

（3）根据前面的讨论，MyLogin 用户控件需要通过事件通知其外部页面。事件发生时，需要把用户名和密码作为事件参数传递给外部页面。因此，需要在项目中添加一个类，描述事件参数。该类从 EventArgs 继承，有 name 和 password 两个属性，代码如下：

```
//登录事件参数
public class MyLoginEventArgs : EventArgs
{
    public string name { get; set; }
    public string password { get; set; }
}
```

（4）在 MyLogin.ascx 用户控件中声明一个事件 myLoginEvent，代码如下：

```
public event EventHandler<MyLoginEventArgs> myLoginEvent; //定义登录事件
```

（5）在 MyLogin.aspx 用户控件“登录”按钮的 Click 事件中，触发自定义的用户事件。代码如下：

```
protected void LoginButton_Click(object sender, EventArgs e)
{
    //如果登录事件附加了事件处理程序
    if (myLoginEvent != null)
    {
        //创建事件参数并设置其值
        MyLoginEventArgs arg = new MyLoginEventArgs();
        arg.name = UserName.Text;
        arg.password = Password.Text;
        myLoginEvent(this, arg);                    //调用事件处理程序
    }
}
```

（6）在 MyLogin.aspx 用户控件中添加两个方法，用于显示和清除错误提示信息。代码如下：

```
//显示登录错误信息
//<param name="message">要显示的错误信息</param>
public void showFailure(string message)
{
```

```
    FailureText.Text = message;
    FailureText.Visible = true;
}
//清除错误信息
public void clearFailure()
{
    FailureText.Text = "";
    FailureText.Visible = false;
}
```

（7）在项目中添加一个 ASP.NET 页面 MyLoginPage.aspx，用于测试用户控件。页面代码如下：

```
<form id="form1" runat="server">
<div>
    <uc1:MyLogin ID="MyLogin1" runat="server" />
</div>
</form>
```

（8）在 MyLoginPage.aspx 页面中，为 MyLogin 用户控件的 myLoginEvent 事件添加事件处理程序，以验证用户名和密码正确性。代码如下：

```
protected void Page_Load(object sender, EventArgs e)
{
    //为登录用户控件的 myLoginEvent 事件添加事件处理程序
    MyLogin1.myLoginEvent += new EventHandler
    <UserControlSample.UserControls.MyLoginEventArgs>
(MyLogin1_myLoginEvent);
}
//登录用户控件的 myLoginEvent 事件处理程序
void MyLogin1_myLoginEvent(object sender,
     UserControlSample.UserControls.MyLoginEventArgs e)
{
    if (e.name == "admin" && e.password == "abcd")      //检测用户名和密码
    {
        Response.Redirect("LoginSuccess.aspx"); //如果正确则转到登录成功页面
    }
    else
    {
        MyLogin1.showFailure("用户名或密码错误");  //如果错误则显示错误提示
    }
}
```

（9）运行 MyLoginPage.aspx 页面，运行界面如图 1.33 所示。



图 1.33　登录控件运行界面

1.6　自定义控件

与用户控件类似，自定义控件也是开发人员自行设计开发的可以在 ASP.NET 页面上使用的控件。但是自定义控件与用户控件相比，也有明显的区别，二者的创建、使用过程都不相同，自定义控件的开发往往比用户控件更难，功能更强大，更能体现开发人员的水平。

1.6.1　自定义控件概述

自定义控件功能强大，其行为类似于 ASP.NET 标准控件。一个设计良好的自定义控件可以复用于各个项目中，完成一定功能，提高开发效率。现在网上有较多优秀的 ASP.NET 自定义控件，如 Infragistics NetAdvantage、Xceed Chart、Teleric、ComponentArt 等。

ASP.NET 自定义控件通常是在一个单独的.NET 项目中，经过编译后生成一个程序集（DLL 文件）。使用自定义控件的开发人员需要在项目中添加对这个程序集的引用，还可以把自定义控件拖动到 Visual Studio 开发环境的工具箱里，然后像使用 ASP.NET 标准控件一样使用自定义控件。对于控件使用者来说，自定义控件与 ASP.NET 标准控件在使用上很相似。

与用户控件相比，自定义控件功能更强大，使用更方便，开发更复杂，可以作为单独的 DLL 文件进行发布。对于一个 ASP.NET 开发人员来说，自定义控件比用户控件更能体现开发人员的思路和水平。国内已经有一些程序员开发出了免费 ASP.NET 自定义控件，并被广泛使用，如用于数据分页的 AspNetPager 控件。

一个 ASP.NET 自定义控件对应于一个类，这个类通常从 System.Web.UI.Control 或 System.Web.UI.WebControls.WebControl 派生。如果自定义控件没有用户界面，那么一般从 Control 类派生，如果自定义控件有用户界面，那么一般从 WebControl 类派生。

1.6.2　创建和使用简单的自定义控件

大多数实用的自定义控件都较为复杂，需要考虑状态保持、回发、事件等各种因素。对于没有控件开发基础的人来说，需要循序渐进地学习和练习。下面将开发一个最简的 HelloWorld 控件，以此来说明创建和使用自定义控件的过程。

【例 1-22】 HelloWorld 控件。

本例将开发一个 HelloWorld 控件，此控件的唯一作用就是在页面上显示“Hello，World”文字。

（1）创建一个 C#类库项目，命名为 CustomControls，注意不要创建成 ASP.NET 项目。正确操作方法如图 1.34 所示。

（2）在项目中添加一个 HelloWorld 类。

（3）本例中，HelloWorld 类需要从 System.Web.UI.WebControls.WebControl 类派生，WebControl 类在 System.Web 程序集中。在默认情况下，类库项目并不包含 System.Web 程序集，需要手动添加对此程序集的引用。操作步骤是在项目上右击，从弹出的快捷菜单中

选择“添加引用”命令。在弹出的“添加引用”对话框中找到 System.Web，单击“确定”按钮，如图 1.35 所示。

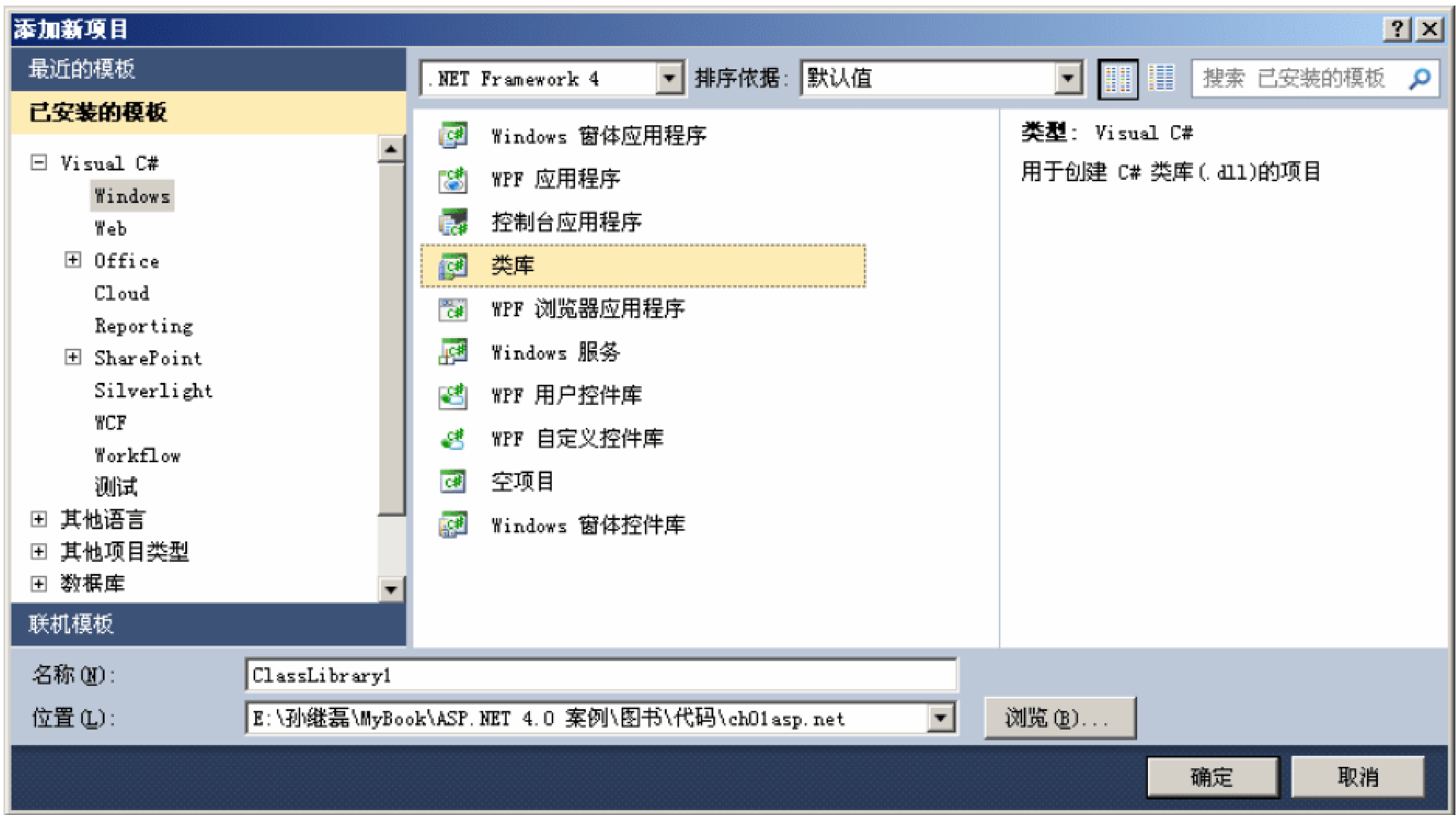


图 1.34　添加类库项目

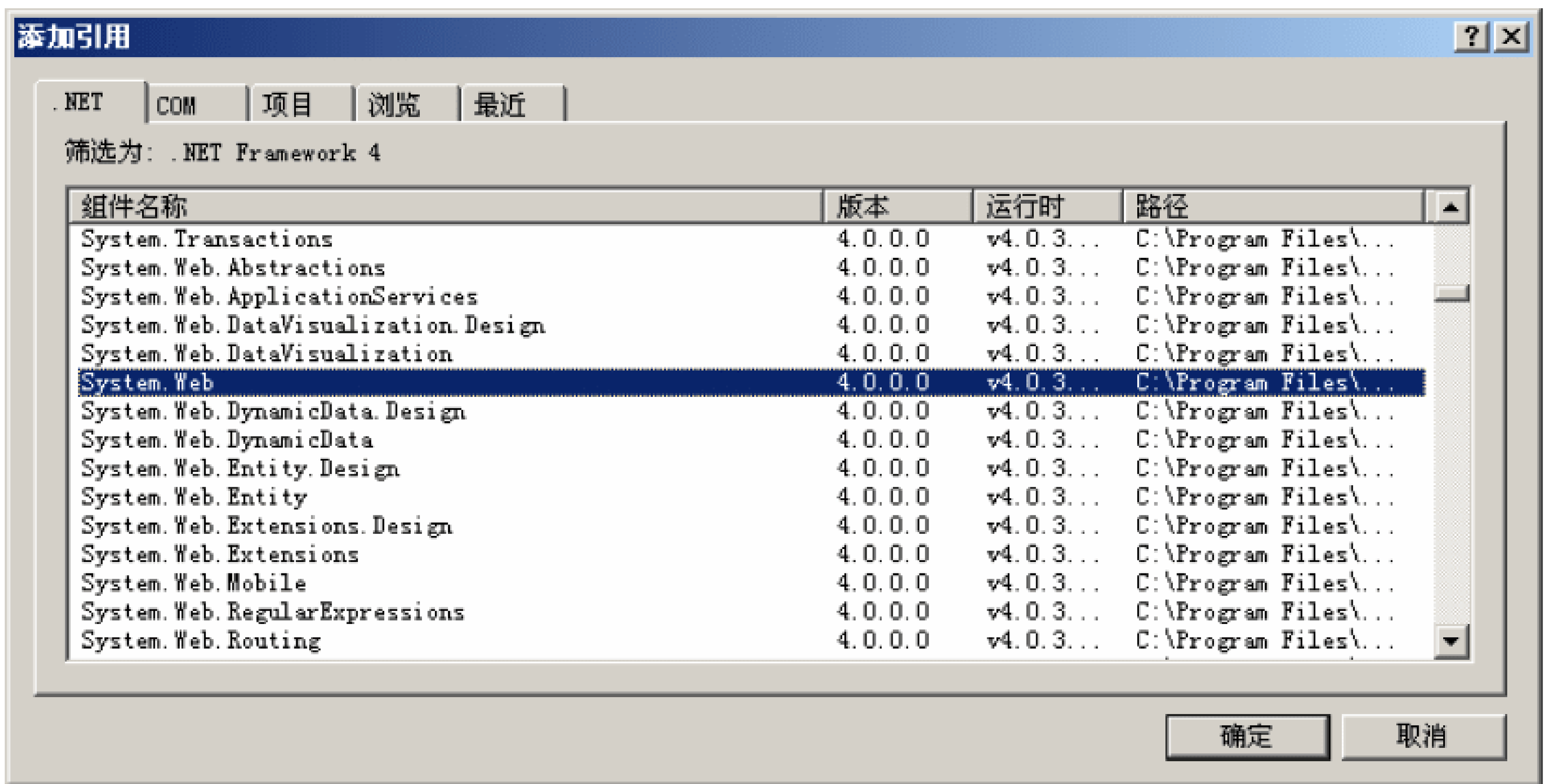


图 1.35　添加对 System.Web 的引用

添加 System.Web 引用后，修改 HellowWorld 类代码，使其从 WebControl 类派生。代码如下：

```
public class HelloWorld:WebControl
{    }
```

（4）要让控件在页面上显示出来，就是让控件在页面上生成 HTML 代码。可以通过重写 WebControl 类的 Render()方法向页面输出任意 HTML 代码。本例需要输出“Hello World”字样，相应代码如下：

```
protected override void Render(System.Web.UI.HtmlTextWriter writer)
{
    writer.Write("<hr/>");
    writer.Write("<span style='font-weight:bold;'>Hello,World.</span>");
    writer.Write("<hr/>");
}
```

（5）编译 CustomControls 项目，则在项目文件夹下的 Bin 目录中生成一个与项目名称相同的 dll 文件。到此为止一个最简单的自定义控件就开发完成了。下面介绍如何在 ASP.NET 项目中使用此控件。

（6）新建一个 ASP.NET Web 应用程序，命名为 ControlTest，此项目是自定义控件的测试项目。

（7）如果刚才所创建的测试项目 ControlTest 与控件项目 CustomControls 在同一解决方案中，则打开 Visual Studio 工具箱，可以看到自动添加了一个 CustomControls 选项卡，选项卡中有一个 HelloWorld 控件。选项卡的名字就是控件项目的名字，选项卡上控件的名字就是控件类的名字，如图 1.36 所示。

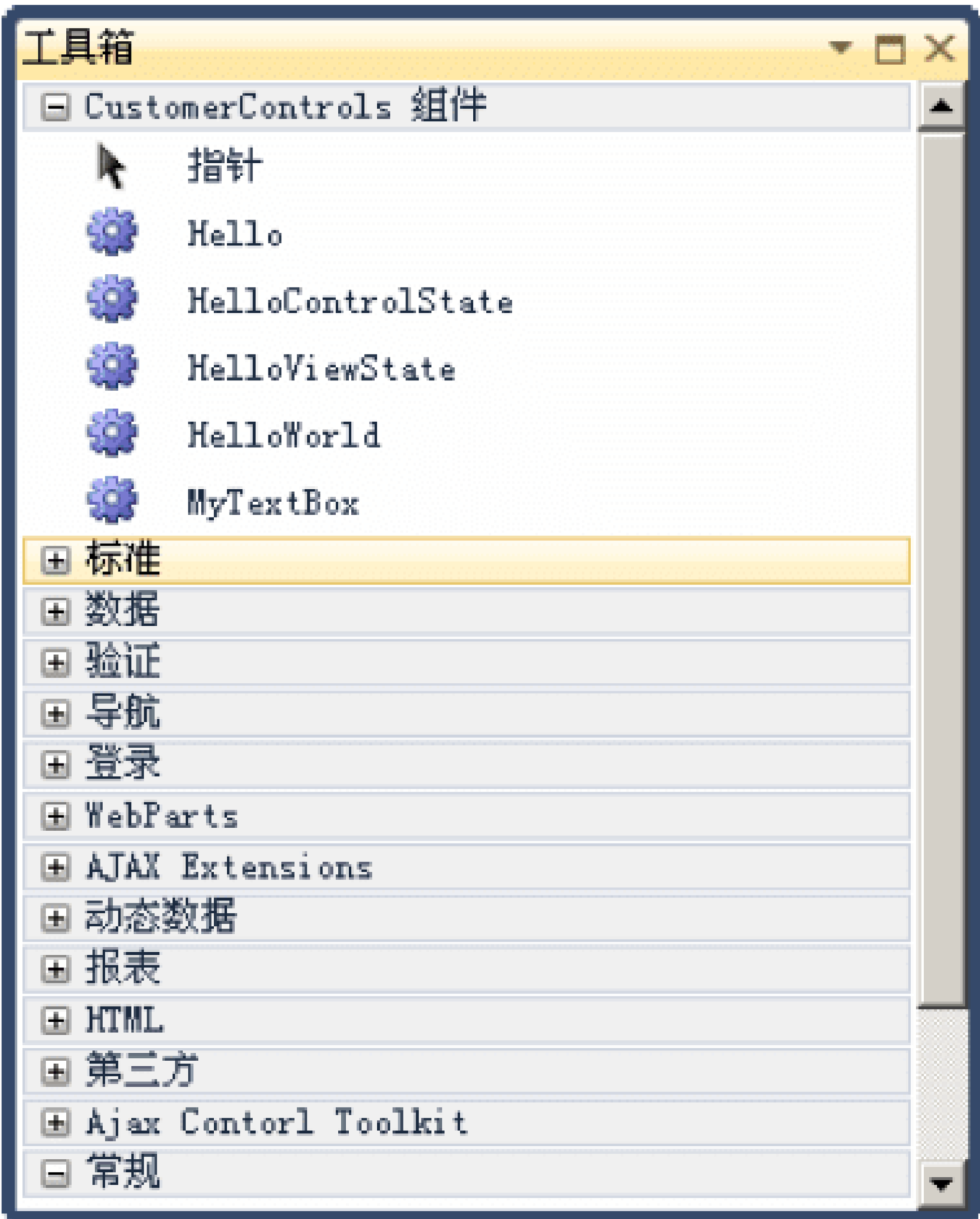


图 1.36　工具箱中的自定义控件

如果刚才所创建的测试项目 ControlTest 与控件项目 CustomControls 不在同一解决方案中，那么需要手工在工具箱中添加自定义控件。具体操作步骤如下。

（a）把控件项目编译后生成的 CustomControls.dll 文件复制到磁盘任意位置。

（b）在 Visual Studio 开发环境工具箱空白处右击，从弹出的快捷菜单中选择“添加选项卡”选项，给新添加的选项卡任意命名，如 MyControl。

（c）打开新添加的选项卡，在选项卡空白处空白处右击，从弹出的快捷菜单中选择“选择项”选项，在打开的对话框中选择刚才所复制的 CustomControls.dll 文件。这样就在工具箱里出现了 HelloWorld 自定义控件。

（8）在 ControlTest 项目中，打开一个页面，从工具箱里把 HelloWorld 控件拖动到页面上，此时页面关键代码如下：

```
<%@  Register  Assembly="CustomerControls"  Namespace="CustomerControls"
TagPrefix="cc1" %>
...
<body>
   <form id="form1" runat="server">
   <div>
   <cc1:HelloWorld runat="server" id="hello1" />
   </div>
   </form>
</body>
```

上述代码中，<%@Register 一行代码注册了 HelloWorld 控件，<cc1:HelloWorld 一行代码使用了该控件。

（9）运行此页面，运行界面如图 1.37 所示。

（10）查看页面 HTML 源码，可以看到如下内容，这些内容正是在 HelloWorld 控件中所写的代码。

```
<hr/>自定义控件示例<br/><span style=不通
'font-weight:bold;'>Hello,World.
</span><hr/>
```

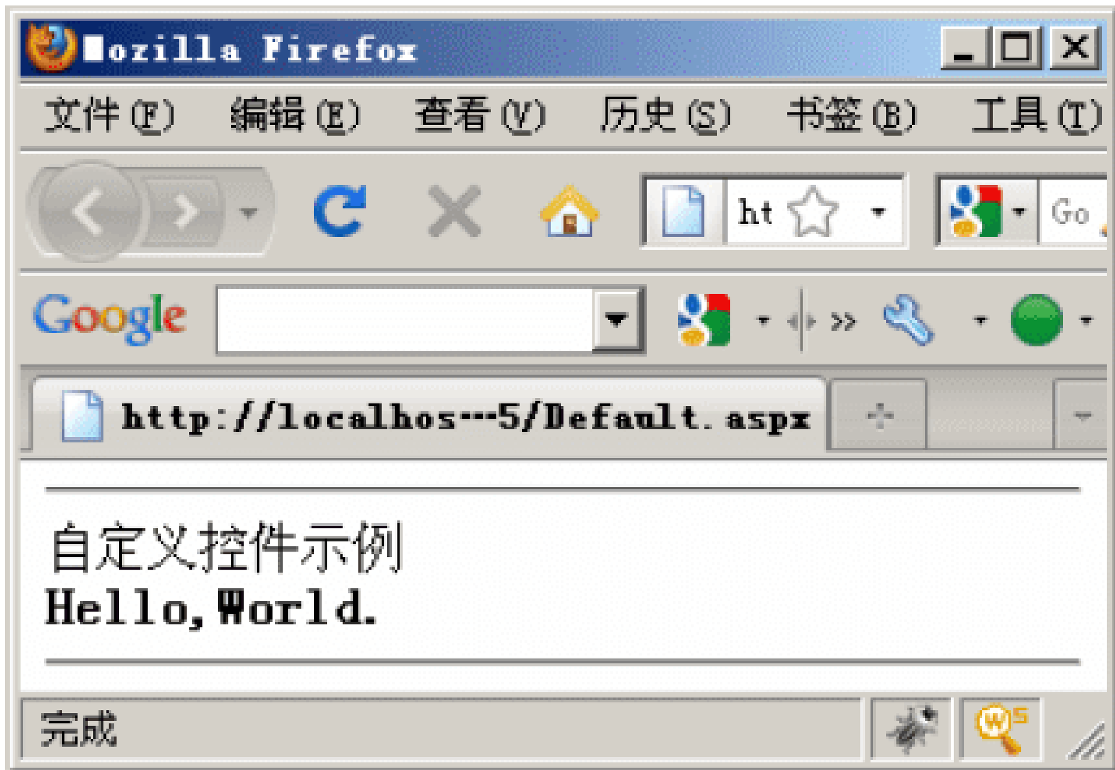


图 1.37　HelloWorld 控件运行界面

1.6.3　添加属性

1.6.2 节所讲的 HelloWorld 控件，只能显示固定的内容，控件使用者无法定义控件的状态，如其中的文字、字体、颜色等。要想实现对控件更加细致的控制，就需要用到自定义属性。下面通过一个例子说明如何为自定义控件添加属性。

【例 1-23】 自定义控件属性。

本例将设计一个简单控件，此控件在页面上输出一条问候语句，如“你好张三”，其中人名、字体颜色都可以设置。

（1）新建一个类库项目 CustomerControls（也可使用例 1-22 所创建的项目）。

（2）在 CustomerControls 项目中添加对 System.Web 和 System.Color 程序集的引用，具体操作步骤与例 1-22 相同。

（3）在 CustomerControls 项目中添加一个继承自 WebControl 类的 Hello 类，代码如下：

```
public class Hello:WebControl
{}
```

（4）从 WebControl 派生的自定义控件默认在页面上呈现为一个 span 元素，即控件的最外层标记为 span，控件内容都在这个标记中。本例将改变这个默认行为，用一个 div 元素显示自定义控件，这需要重写 TagKey 属性，如下代码所示。

```
//将自定义控件的 HTML 标记设置为 div（默认为 span）
protected override HtmlTextWriterTag TagKey
{
    get
    {
        return HtmlTextWriterTag.Div;
    }
}
```

（5）在 Hello 类中，添加 3 个属性，分别描述姓名和两种颜色。代码如下：

```
public string name { get; set; }            //你好后面的姓名
public Color helloColor { get; set; }       //“你好”的颜色
public Color nameColor { get; set; }        //姓名的颜色
```

（6）在 Hello 类中重写 RenderContents()方法，按照指定颜色输出问候语。代码如下：

```
//输出控件内容
protected override void RenderContents(HtmlTextWriter writer)
{
    string c1=ColorTranslator.ToHtml(helloColor);
                                            //将.NET 的颜色转换为 HTML 形式
   //添加颜色样式，此语句将生成以下代码： style="color:颜色"
    writer.AddAttribute(HtmlTextWriterAttribute.Style, "color:" + c1);
    writer.RenderBeginTag(HtmlTextWriterTag.Span);
                                            //输出一个 span 开始标记<span>
    writer.Write("你好");                   //输出 你好
    writer.RenderEndTag();                  //结束前一个 HTML 标记，即 span
    string c2=ColorTranslator.ToHtml(nameColor);
    //以下语句以特定颜色在 span 中输出姓名
    writer.AddAttribute(HtmlTextWriterAttribute.Style, "color:" + c2);
```

```
    writer.RenderBeginTag(HtmlTextWriterTag.Span);
    writer.Write(name);
    writer.RenderEndTag();
}
```

（7）编译 CustomerControls 项目以生成 dll 文件。

（8）新建一个 ASP.NET Web 应用程序 TestControl 作为控件的测试项目（也可以使用例 1-22 所建的项目）。

（9）在 TestControl 项目中引用 CustomerControls.dll 文件，并把控件放到工具箱中。具体操作步骤与例 1-22 所述相同。

（10）在 TestControls 项目中添加一个页面，将 Hello 控件拖到页面上，并设置控件的 3 个属性。页面关键代码如下：

```
<%--注册自定义控件--%>
<%@ Register Assembly="CustomerControls" Namespace="CustomerControls"
TagPrefix="cc1" %>
...
<body >
    <form id="form1" runat="server"><div>
    <%--放置一个自定义控件 Hello，并设置三个属性--%>
    <cc1:Hello runat="server" id="hello2" name="赵钱孙" helloColor="Blue"
nameColor="Green"/>
    </div></form>
</body>
```

（11）运行上一步所添加的页面，可以看到，在浏览器中以指定的颜色显示了问候语。运行界面如图 1.38 所示。

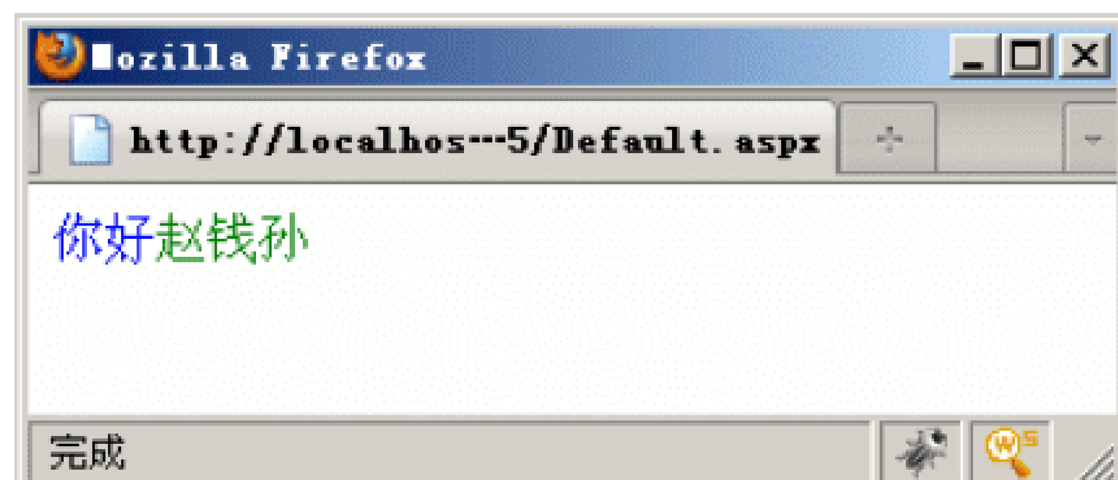


图 1.38　自定义控件属性示例

1.6.4　状态保持概述

HTTP 协议是一种无状态协议，基于 HTTP 协议的 Web 应用程序默认情况下也是无状态的，这给 Web 应用的开发造成不便。各种 Web 开发工具（包括 ASP.NET）都采用一定的方式实现状态保持，如 Session 等。在开发自定义控件时，如果考虑不到实现状态保持功能，那么控件的运行就会出现意想不到的错误，例如，把某一控件的 Text 属性设置为“测试”，经过页面回发以后，这个控件的 Text 属性就会恢复成原来的值。下面通过一个例子来演示这种情况。

【例 1-24】 有状态和无状态。

本例对比 ASP.NET 标准控件（以 Label 为例）和 1.6.3 节开发的自定义控件 Hello 在状态保持方面的区别，以说明状态保持的概念和作用。

（1）创建一个 ASP.NET Web 应用程序。

（2）添加一个新的页面 Stateless.aspx，在页面上放置一个 Label 及一个自定义控件 Hello，然后再放置一个 TextBox 和两个 Button，页面关键代码如下：

```
<form id="form1" runat="server">
<div>
    ASP.NET 标准控件（以下面的 Label 为例）能够保持状态<br />
    <asp:Label ID="Label1" runat="server" Text="Label" BorderStyle=
```

```
    "Solid"></asp:Label><br />
自定义控件 Hello 不能保持状态<br />
    <cc1:Hello runat="server" ID="hello" name="初始值" helloColor="Blue"
    nameColor="Green" BorderStyle="Solid"/>
    输入新值：<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
    <asp:Button ID="Button1" runat="server" Text="设置新值" onclick=
    "Button1_Click" />
    <asp:Button ID="Button2" runat="server" Text="回发" />
</div>
</form>
```

（3）当单击“设置新值”Button 时，将会重新设置 Label 和 Hello 控件上的文本，如下代码所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    Label1.Text = TextBox1.Text;  //为 Label 控件设置新的文本
    hello.name = TextBox1.Text;   //为 Hello 控件设置新的姓名
}
```

（4）页面上“回发”Button 的作用是引起页面回发，从而检验控件能否保持状态。由于 ASP.NET 的服务器端 Button 控件默认情况下将引起回发，所以不需要为此 Button 编写事件处理程序。

（5）运行 Stateless.aspx 页面，在文本框中输入新的文字，单击“设置新值”按钮，可以看到，Label 控件和 Hello 控件的文字都发生了变化。此时，再单击“回发”按钮引起一次页面回发过程，当页面重新加载后，可以看到，Label 文本仍然是新的文字，而 Hello 上的文本已经变成了原来的文本，即 Label 能够记忆并且保持其文本内容，而 Hello 则不能，如图 1.39 所示。

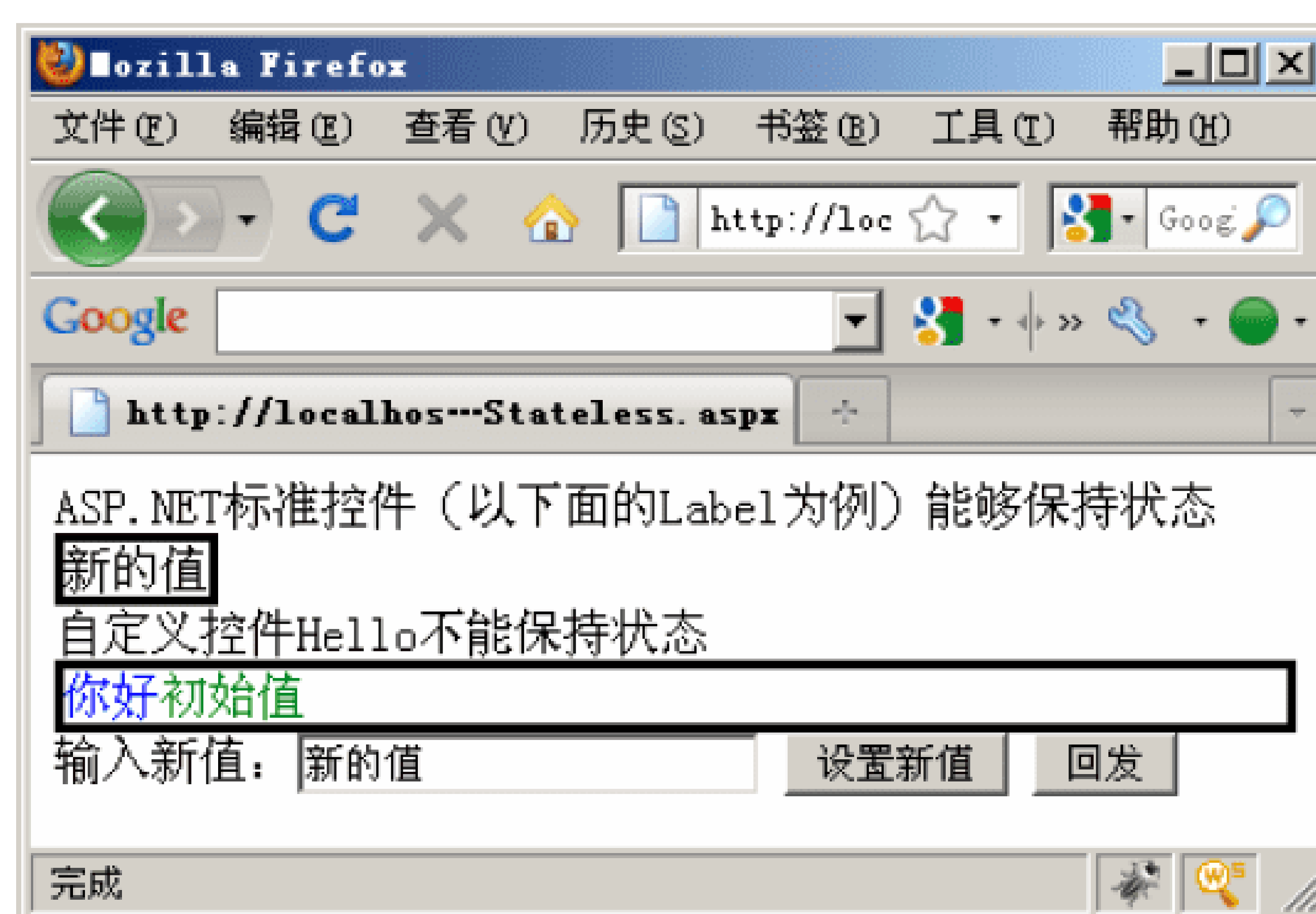


图 1.39　有状态和无状态对比示例

1.6.5　视图状态 ViewState

ASP.NET 标准控件主要采用两种机制实现状态保持功能：视图状态（ViewState）和控件状态（ControlState）。在开发自定义控件时通常也使用这两种机制。

ViewState 属性提供一个字典对象，用于在对同一页的多个请求之间保留值。这是页用来在往返行程之间保留页和控件属性值的默认方法。在处理页时，页和控件的当前状态

会散列为一个字符串，并在页中保存为一个隐藏域或多个隐藏域（如果存储在 ViewState 属性中的数据量超过了 MaxPageStateFieldLength 属性中的指定值）。当将页回发到服务器时，页会在页初始化阶段分析视图状态字符串，并还原页中的属性信息。

在浏览器中浏览一个 ASP.NET 页面时，从浏览器菜单中查看页面的 HTML 源代码，会看到在 form 中有一个名称为“__VIEWSTATE”的隐藏域（Hidden），其中的内容是一些不可读的字符，这就是被编码后的视图状态。下面是一个页面中的 ViewState 内容。

```
<input  type="hidden"  name="__VIEWSTATE"  id="__VIEWSTATE"  value=
"/wEPDwUKMTc2ODM1Mzg1OA9kFgICAw9kFgICAQ8PFgIeBFRleHQFCHJlZHNmZHNmZGRkta
RgeiFCG54vjJxWMYiYGkvCGbY=" />
```

ViewState 是一个 System.Web.UI.StateBag 类的实例，可以通过 System.Web.UI.Control 类的 ViewState 属性得到一个控件或者页面的 ViewState。ViewState 属性声明如下：

```
protected virtual StateBag ViewState { get; }
```

StateBag 是一个类似于字典 Dictionary 的类，可以通过一个字符类型的关键字为索引得到与之相对应的内容。如果没有与关键字匹配的内容，则返回 null。

【例 1-25】 控件的视图状态。

本例演示通过视图状态机制保持控件的属性。

（1）创建一个类库项目。

（2）在项目中添加一个类 HelloViewState，此类功能与例 1-23 中的 Hello 控件类似，也是根据指定的颜色和姓名显示一条问候信息。所不同的是，这个控件使用了 ViewState 保持控件属性。HelloViewState 类代码如下：

```
//自定义控件示例，通过 ViewState 保持状态
public class HelloViewState : WebControl
{
    #region 自定义属性，通过 ViewState 保存这些属性
    //下面 3 个常量为 3 个属性在 ViewState 中的键值
    private const string ViewKeyName = "nameviewstate";
    private const string ViewKeyColor1 = "hellocolor";
    private const string ViewKeyColor2 = "namecolor";
    //你好后面的姓名
    public string name
    {
        get
        {
            return ViewState[ViewKeyName] as string;
        }
        set
        {
            ViewState[ViewKeyName] = value;
        }
    }
    //“你好”的颜色
    public Color helloColor
    {
        get
        {
            if (ViewState[ViewKeyColor1]==null)
                return Color.Green;
            return (Color)ViewState[ViewKeyColor1];
        }
```

```
        set
        {
            ViewState[ViewKeyColor1] = value;
        }
    }
    //“姓名”的颜色
    public Color nameColor
    {
        get
        {
            if (ViewState[ViewKeyColor2] == null)
                return Color.Green;
            return (Color)ViewState[ViewKeyColor2];
        }
        set
        {
            ViewState[ViewKeyColor2] = value;
        }
    }
    #endregion
    //将自定义控件的 HTML 标记设置为 div（默认为 span）
    protected override HtmlTextWriterTag TagKey
    {
        get
        {
            return HtmlTextWriterTag.Div;
        }
    }
protected override void RenderContents(HtmlTextWriter writer)
{
    string c1 = ColorTranslator.ToHtml(helloColor);
                                        //将.NET 的颜色转换为 HTML 形式
    //添加颜色样式，此语句将生成以下代码：  style="color:颜色"
    writer.AddAttribute(HtmlTextWriterAttribute.Style, "color:" + c1);
    writer.RenderBeginTag(HtmlTextWriterTag.Span);
                                        //输出一个 span 开始标记<span>
    writer.Write("你好");                //输出 你好
    writer.RenderEndTag();              //结束前一个 HTML 标记，即 span
    //以下语句以特定颜色在 span 中输出姓名
    string c2 = ColorTranslator.ToHtml(nameColor);
    writer.AddAttribute(HtmlTextWriterAttribute.Style, "color:" + c2);
    writer.RenderBeginTag(HtmlTextWriterTag.Span);
    writer.Write(name);
    writer.RenderEndTag();
}
```

（3）新建一个 ASP.NET 项目，在项目中添加一个页面 ControlViewState.aspx 以测试 HelloViewState 控件。页面关键代码如下：

```
<form id="form1" runat="server">
<div>
<cc1:HelloViewState runat="server" ID="control1" name="初始值" helloColor=
"Green" nameColor="Blue" />
   输入新值：<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
   <asp:Button ID="Button1" runat="server" Text="设置新值" onclick=
   "Button1_Click" />
   <asp:Button ID="Button2" runat="server" Text="回发" />
</div>
</form>
```

（4）ControlViewState.aspx.aspx 页面后台代码如下：

```
protected void Button1 Click(object sender, EventArgs e)
{
    control1.name = TextBox1.Text;
}
```

（5）运行 ControlViewState.aspx.aspx 页面，设置新的姓名并回发页面，可以看到，所设置的新姓名能够保存在控件中。运行界面如图 1.40 所示。

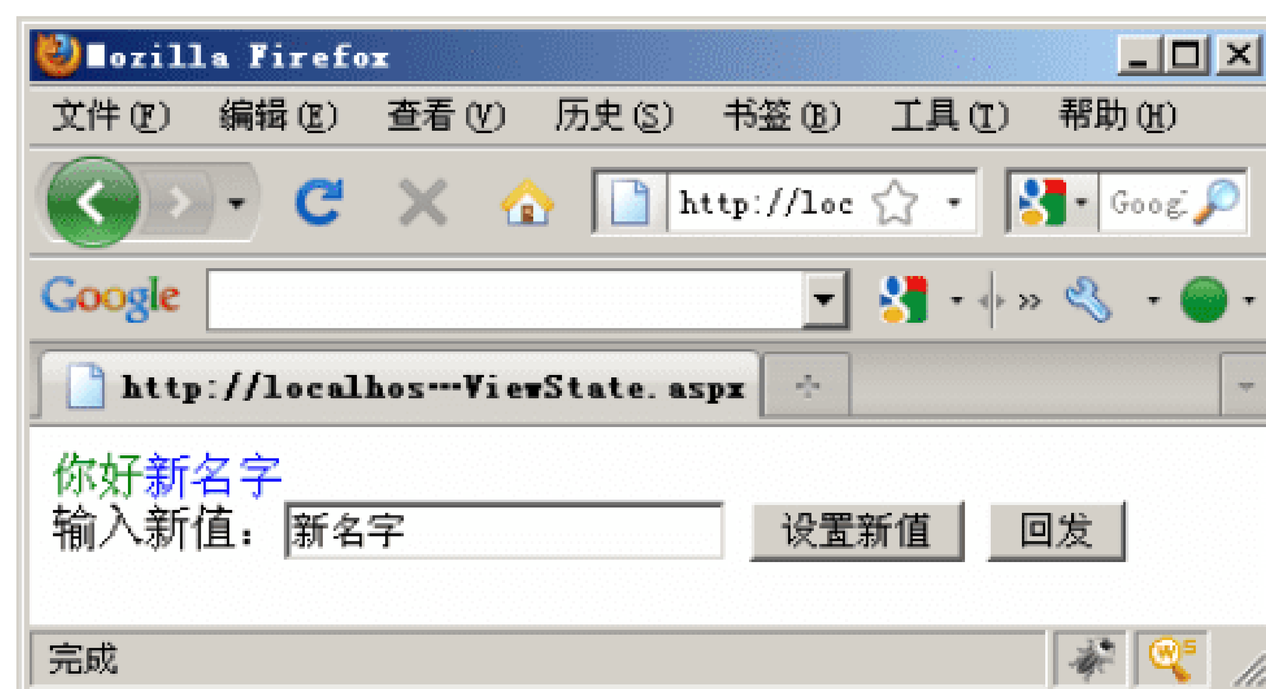


图 1.40　ViewState 保持状态

1.6.6　控件状态 ControlState

默认情况下，ASP.NET 服务器端控件都会用 ViewState 进行状态保持，这样就会在 ASP.NET 页面中产生很大的隐藏域，影响服务器性能及页面加载速度。针对这种情况，ASP.NET 允许开发人员自己决定是否使用 ViewState，以避免产生不必要的垃圾数据。开发人员可以在页面的<%@Page 中启用或者禁用 ViewState，如下代码所示。

```
<!--下面这行代码　启用　了 ViewState-->
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Page1.aspx.cs"
Inherits= "Test.Page1" EnableViewState="true"%>
<!--下面这行代码　禁用　了 ViewState-->
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Page1.aspx.cs"
Inherits= "Test.Page1" EnableViewState="false"%>
```

如果一个控件用 ViewState 来进行状态保持，而控件所在的页面禁用了 ViewState 被，那么控件的状态保持机制就不能正常工作。这时候，需要另外一种状态保持机制——控件状态。控件状态的工作方式与视图状态类似，也是把状态数据散列为一个字符串，并把此字符串保存到页面的隐藏域中。与视图状态不同的是，控件状态不能被禁用，始终有效。控件状态是专为存储控件的重要数据（如一个页面控件的页数）而设计的，回发时必须用到这些数据才能使控件正常工作（即便禁用视图状态也不受影响）。

【例 1-26】 控件状态。

本例演示控件状态在自定义控件中的使用，并将其与视图状态对比。

（1）创建一个类库项目，在项目中添加一个继承自 WebControl 的类 HelloControlState。代码如下：

```
// 自定义控件示例，通过 ControlState 保持状态
public class HelloControlState : WebControl
{    }
```

（2）在 HelloControlState 类中添加 3 个属性，注意与例 1-25 区别，这 3 个属性并不保

存在 ViewState 中。代码如下：

```
#region 自定义属性，通过 ControlState 保存这些属性
public string name { get; set; }
public Color helloColor { get; set; }
public Color nameColor { get; set; }
#endregion
```

（3）在 HelloControlState 类中添加一个内部类 HelloProperty，这个类用于向控件状态中保存和恢复控件的属性。代码如下：

```
//内部类，用于封装控件的属性，以便于保存和读取
//此类必须声明为可序列化
[Serializable]
class HelloProperty
{
    public string name { get; set; }
    public Color helloColor { get; set; }
    public Color nameColor { get; set; }
}
```

提示： 要保存到 ControlState 中的属性类型必须是可序列化的。.NET 中的简单类如 string、int、double 都是可序列化的，如果是自定义类，默认情况下都是不可序列化的，可以在类前面加上[Serializable]明确指定类可序列化。

（4）在 HelloControlState 类中重写 SaveControlState()方法，将控件属性保存到 ControlState 中。代码如下：

```
//将控件属性保存到 ControlState
//<returns>被保存的 ControlState</returns>
protected override object SaveControlState()
{
    HelloProperty property = new HelloProperty();  //创建一个属性类
    //把控件属性保存到属性类的各个字段中
    property.name = this.name;
    property.helloColor = this.helloColor;
    property.nameColor = this.nameColor;
    object o=base.SaveControlState();  //调用基类的 SaveControlState()方法
    if (o != null)
    {
        //如果基类保存的 ControlState 不为空
        //则将基类的 ControlState 与该控件自定义的属性一起返回
        return new Pair(o, property);
    }
    else
    {
        //如果基类保存的 ControlState 为空，则只需要返回该控件自定义的属性
        return property;
    }
}
```

（5）在 HelloControlState 类中重写 LoadControlState()方法，从保存的 ControlState 中恢复控件属性。代码如下：

```
/// <summary>
/// 从 ControlState 中恢复控件属性
```

```
/// </summary>
/// <param name="savedState">被保存的 ControlState</param>
/// 这个方法是 SaveControlState 方法的逆过程。读取数据时，应按照与保存数据相一致的格式
protected override void LoadControlState(object savedState)
{
    if (savedState == null) return;             //没有被保存的数据则返回
    //如果被保存的数据为 Pair 类，则说明 Pair 的前一元素为基类保存的数据
    //后一元素为当前类保存的数据（即 HelloProperty）
    if (savedState is Pair)
    {
        Pair pair = savedState as Pair;
        base.LoadControlState(pair.First);      //恢复基类保存的控件状态
      HelloProperty property = pair.Second as HelloProperty;
                                                //得到当前类保存的控件状态
        if (property == null) return;
        //根据读取的控件状态数据恢复控件属性
        this.name = property.name;
        this.helloColor = property.helloColor;
        this.nameColor = property.nameColor;
    }
    else if (savedState is HelloProperty)
    {
        //如果被保存的数据是 HelloProperty 类型，则直接恢复控件属性
        HelloProperty property = savedState as HelloProperty;
        if (property == null) return;
        this.name = property.name;
        this.helloColor = property.helloColor;
        this.nameColor = property.nameColor;
    }
    else
    {
    //如果被保存的数据既不是 Pair，也不是 HelloProperty，那么调用基类的方法恢复控件
    状态
        base.LoadControlState(savedState);
    }
}
```

提示： 如果自定义控件用控件状态保存数据，需要重写 SaveControlState()方法和 LoadControlState()方法。这两个方法是互逆的过程。SaveControlState 按照一定的格式保存数据，而 LoadControlState 按照相同的格式把保存的数据读出。

（6）在 HelloControlState 类中重写 OnInit()方法，将控件注册为拥有 ControlState。代码如下：

```
protected override void OnInit(EventArgs e)
{
   base.OnInit(e);
   Page.RegisterRequiresControlState(this);
                                        //将此控件注册为具有持久性控件状态
}
```

提示： 当自定义控件用 ControlState 保存控件数据时，必须把控件注册为具有拥有 ControlState。只有经过注册的控件，ASP.NET 才会调用其 LoadControlState()和 SaveControlState()方法。否则，即使重写了 LoadControlState()和 SaveControlState()方法，这些方法也不会被执行。

（7）在 HelloControlState 类中重写 RenderContents()方法，输出控件内容。代码如下：

```
protected override void RenderContents(HtmlTextWriter writer)
{
    string c1 = ColorTranslator.ToHtml(helloColor);
                                        //将.NET 的颜色转换为 HTML 形式
    //添加颜色样式，此语句将生成以下代码： style="color:颜色"
    writer.AddAttribute(HtmlTextWriterAttribute.Style, "color:" + c1);
    writer.RenderBeginTag(HtmlTextWriterTag.Span);
                                        //输出一个 span 开始标记<span>
    writer.Write("你好");               //输出 你好
    writer.RenderEndTag();              //结束前一个 HTML 标记，即 span
    //以下语句以特定颜色在 span 中输出姓名
    string c2 = ColorTranslator.ToHtml(nameColor);
    writer.AddAttribute(HtmlTextWriterAttribute.Style, "color:" + c2);
    writer.RenderBeginTag(HtmlTextWriterTag.Span);
    writer.Write(name);
    writer.RenderEndTag();
}
```

（8）新建一个 ASP.NET 项目，在项目中添加一个页面 ViewStateControlState.aspx 以测试 HelloControlState 控件。在页面上放置一个本例创建的 HelloControlState 控件和一个例 1-25 创建的 HelloViewState 控件，禁用页面的 ViewState。页面代码如下：

```
<form id="form1" runat="server">
<div>
   下面这个控件用 ViewState 保存状态<br />
   <cc1:HelloViewState runat="server" ID="control1" name="初始值" hello
   Color="Green" nameColor="Blue" /><br />
   下面这个控件用 ControlState 保存状态<br />
   <cc1:HelloControlState runat="server" ID="control2" name="初始值" hello
   Color="Green" nameColor="Blue" /><br />
 输入新值：<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
   <asp:Button ID="Button1" runat="server" Text="设置新值" onclick=
   "Button1 Click" />
   <asp:Button ID="Button2" runat="server" Text="回发" />
</div>
</form>
```

（9）为 ViewStateControlState.aspx 页面上的 Button 编写事件处理程序，代码如下：

```
protected void Button1_Click(object
sender, EventArgs e)
{
   control1.name = TextBox1.Text;
   control2.name = TextBox1.Text;
}
```

（10）运行 ViewStateControlState.aspx 页面，输入新值以进行测试。可以看到，在禁用了 ViewState 的页面中，保存在 ViewState 中的属性被丢失，而保存在 ControlState 中的属性工作正常。程序运行结果如图 1.41 所示。

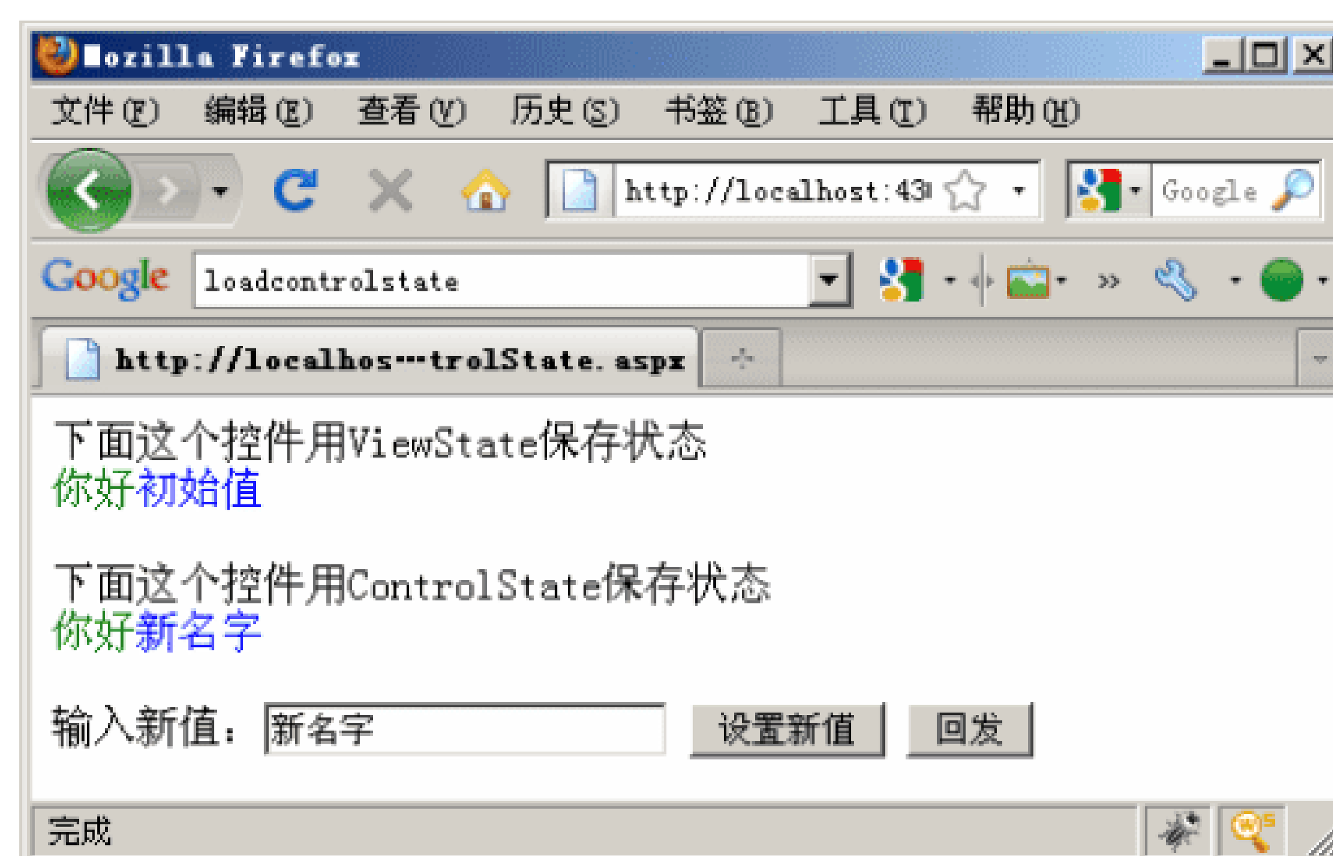


图 1.41　控件状态与视图状态

1.6.7　回发数据和事件

ASP.NET 的页面呈现为 HTML 时会生成一个 form，页面回发时会把 form 及 form 上用户输入的数据提交给服务器。ASP.NET 标准服务器端控件可以处理这些回发的数据，而自定义控件需要编写相应代码才能完成这些工作。

如果一个自定义控件需要处理浏览器的回发数据，则需要继承 IPostBackDataHandler 接口。IPostBackDataHandler 接口以下有两个方法。

- ❑ LoadPostData 方法：此方法接收从浏览器回发的数据。
- ❑ RaisePostDataChangedEvent 方法：此方法产生数据更改事件。

【例 1-27】 简单的 TextBox 控件。

本例将模仿 ASP.NET 标准控件中的 TextBox 编写一个 MyTextBox 控件，这个控件允许用户在其中输入文本，并可以处理回发的这些文本。如果新输入的文本与原来文本不同，则触发一个事件。

（1）新建一个类库项目 CustomControls。

（2）在 CustomControls 添加一个 MyTextBox 类，该类从 WebControl 类派生，并实现 IPostBackDataHandler 接口。

```
public class MyTextBox:WebControl,IPostBackDataHandler
{}
```

（3）MyTextBox 控件是一个文本控件，控件标记应为 input，重写 TagKey 属性以实现此功能。代码如下：

```
//控件的 HTML 标记为 input
protected override HtmlTextWriterTag TagKey
{
    get
    {
        return HtmlTextWriterTag.Input;
    }
}
```

（4）为 MyTextBox 控件添加一个 text 属性，该属性保存在 ViewState 中。代码如下：

```
private const string ViewStateKey="ViewStateText";
//控件的 text 属性
public string text
{
    get
    {
        if (ViewState[ViewStateKey] == null)
            return null;
        else
            return ViewState[ViewStateKey] as string;
    }
    set
    {
        ViewState[ViewStateKey] = value;
    }
}
```

（5）为 MyTextBox 控件的 HTML 标记添加适当的属性，代码如下：

```
//在呈现控件前为控件添加属性
protected override void AddAttributesToRender(HtmlTextWriter writer)
{
    writer.AddAttribute(HtmlTextWriterAttribute.Type, "text");
    //添加属性：type="text"
    //添加属性：style="border:solid 1px blue"
    writer.AddStyleAttribute(HtmlTextWriterStyle.BorderStyle,"solid" );
    writer.AddStyleAttribute(HtmlTextWriterStyle.BorderWidth, "1px");
    writer.AddStyleAttribute(HtmlTextWriterStyle.BorderColor, "blue");
    //添加属性 name=<控件名称>
    writer.AddAttribute(HtmlTextWriterAttribute.Name, this.UniqueID);
    writer.AddAttribute(HtmlTextWriterAttribute.Value,text);
    //添加属性 value=text 值
    base.AddAttributesToRender(writer);
}
```

（6）将 MyTextBox 控件注册为需要处理回发，代码如下：

```
protected override void OnInit(EventArgs e)
{
    base.OnInit(e);
    Page.RegisterRequiresPostBack(this);        //将控件注册为处理回发
}
```

（7）为 MyTextBox 控件添加一个 textChanged 事件：

```
//定义事件 myTextChanged，当文本属性时触发
public event EventHandler<EventArgs> myTextChanged;
```

（8）实现 IPostBackDataHandler 接口的 LoadPostData 方法以处理回发数据。代码如下：

```
//处理回发数据
public   bool   LoadPostData(string   postDataKey,   System.Collections.
Specialized.NameValueCollection postCollection)
{
    string newText = postCollection[postDataKey];      //取得控件中的新文本
    //如果新文本与旧文本相同，则不触发 Changed 事件，否则触发 Changed 事件
    if (newText == text)
        return false;
    else
    {
        text = newText;
        return true;
    }
}
```

（9）实现 IPostBackDataHandler 接口的 RaisePostDataChanged()方法以触发数据更改事件。代码如下：

```
//触发数据更改事件
public void RaisePostDataChangedEvent()
{
    OnTextChanged(EventArgs.Empty);
}
protected virtual void OnTextChanged(EventArgs e)
{
    if (myTextChanged != null)
```

```
        myTextChanged(this, e);
}
```

（10）新建一个 ASP.NET 项目，在项目中添加一个页面 MyTextBoxPage.aspx 以测试 MyTextBox 控件。页面代码如下：

```
<form id="form1" runat="server"> <div>
<cc1:MyTextBox runat="server" ID="textbox1"
       onmytextchanged="textbox1_myTextChanged" />
    <asp:Button ID="Button1" runat="server" Text="回发" onclick="Button1_
    Click" />
    <br />
    <asp:Label ID="Label1" runat="server" Text="输出结果"></asp:Label>
</div>
</form>
```

（11）在 MyTextBoxPage.aspx 页面上为 MyTextBox 控件的 myTextChanged 事件编写如下代码：

```
protected void textbox1 myTextChanged(object sender, EventArgs e)
{
    Label1.Text = "myTextBox 文本发生改变，新的文本为"+textbox1.text;
}
```

（12）在浏览器中运行 MyTextBoxPage.aspx 页面，在 MyTextBox 控件中输入文本，单击“回发”按钮，则控件中的文本可以回发到服务器，并触发 myTextChanged 事件。运行界面如图 1.42 所示。

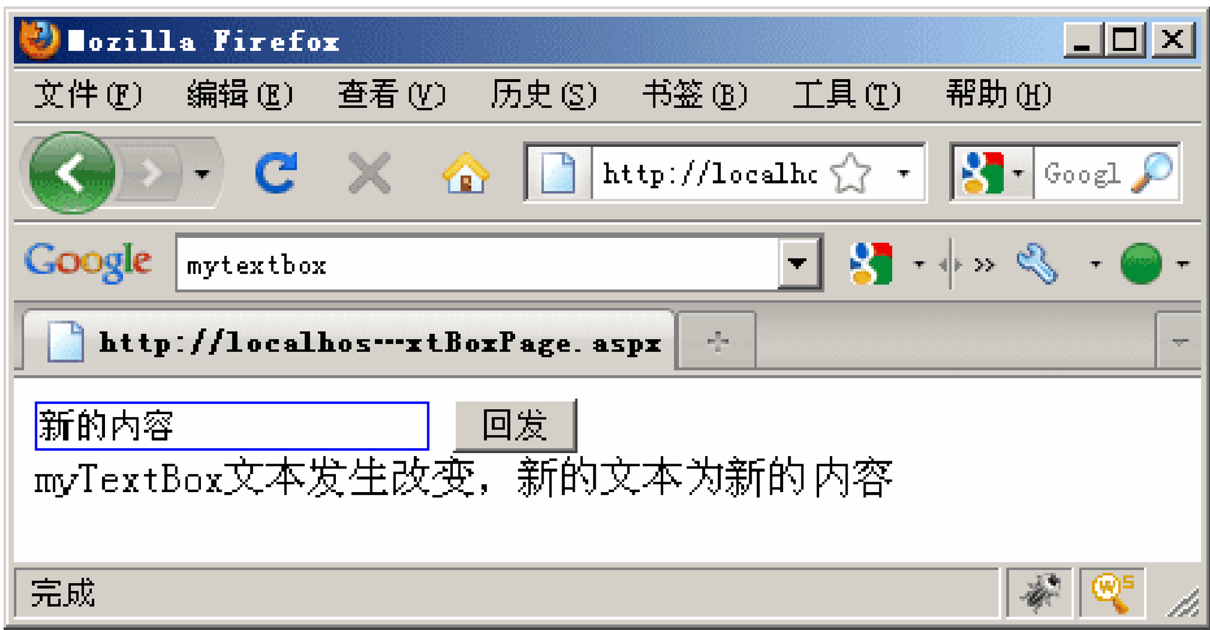


图 1.42　MyTextBox 运行界面

1.7　小　　结

本章介绍了 ASP.NET 编程的一些高级技术。首先分析了 ASP.NET 页面生命周期和事件模型，使读者从一定深度上理解 ASP.NET 技术，接着介绍了母版、主题、用户控件和自定义控件，其中自定义控件部分是本章的难点。

第 2 章　ADO.NET 数据库访问技术

ADO.NET 是.NET 架构中用于访问数据库的组件，ADO.NET 组件主要存在于 System.Data 及其子命令空间中。通过 ADO.NET，可以使用相同的方式访问和操作各种类型的数据库，如 Oracle、SQL Server 等。本章将主要以访问 SQL Server 数据库为例说明 ADO.NET 的功能和使用方法。

2.1　ADO.NET 概述

ADO.NET 通过数据处理将数据访问分解为多个可以单独使用或一前一后使用的不连续组件。ADO.NET 包含用于连接到数据库、执行命令和检索结果的.NET Framework 数据提供程序。ADO.NET 用于访问和操作数据的两个主要组件是.NET Framework 数据提供程序和 DataSet，如图 2.1 所示。

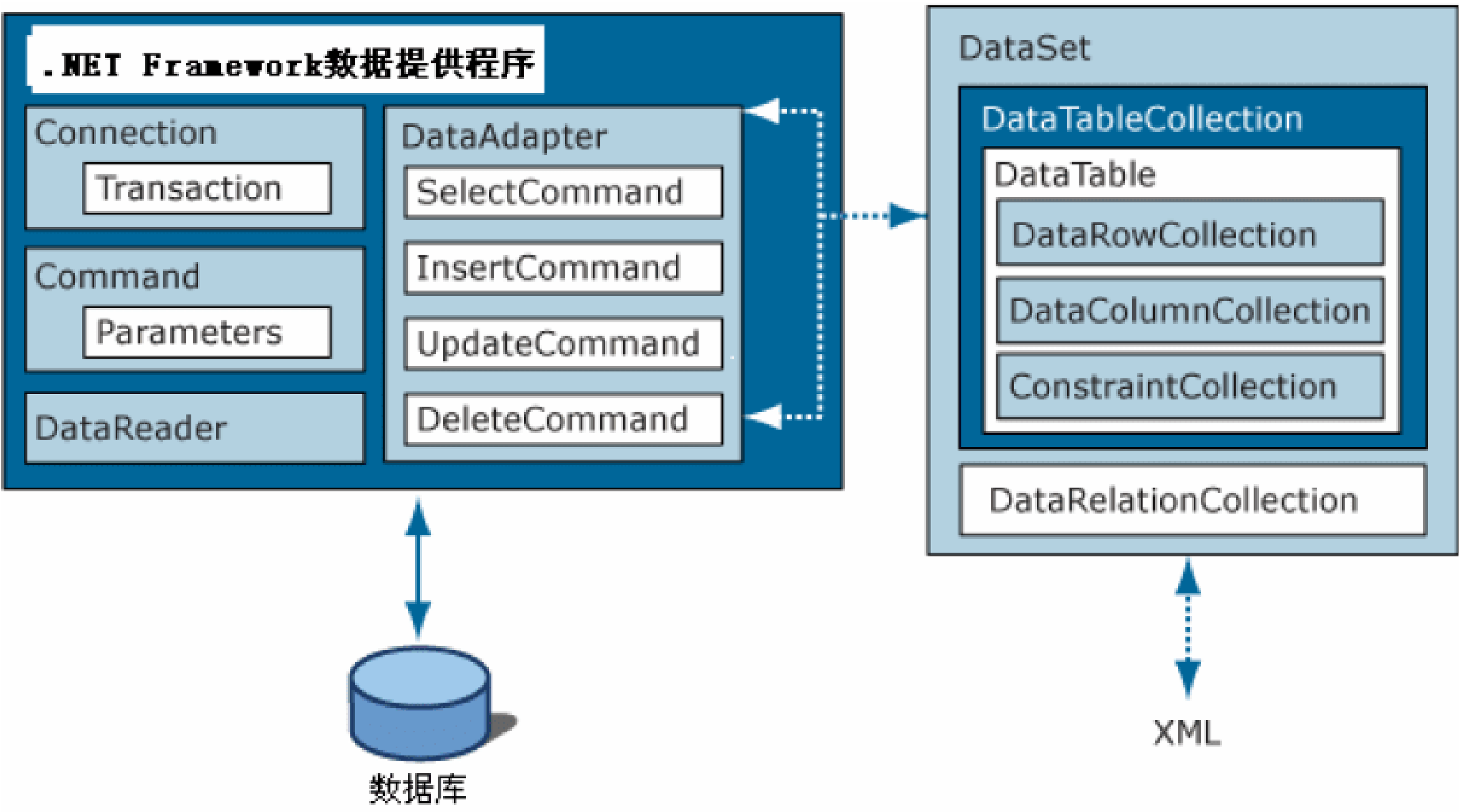


图 2.1　ADO.NET 结构

.NET Framework 数据提供程序是专门为数据操作及快速、只进、只读访问数据而设计的组件。Connection 对象提供到数据源的连接。使用 Command 对象可以访问用于返回数据、修改数据、运行存储过程，以及发送或检索参数信息的数据库命令。DataReader 可从数据源提供高性能的数据流。DataAdapter 在 DataSet 对象和数据源之间起到桥梁作用。DataAdapter 使用 Command 对象在数据源中执行 SQL 命令以向 DataSet 中加载数据，并将对 DataSet 中数据的更改协调回数据源。

ADO.NET DataSet 是专门为独立于任何数据源的数据访问而设计的，可以用于多种不

同的数据源，用于 XML 数据，或用于管理应用程序本地的数据。DataSet 包含一个或多个 DataTable 对象的集合，这些对象由数据行和数据列及有关 DataTable 对象中数据的主键、外键、约束和关系信息组成。

2.2　连接数据库

数据库与应用程序（如 ASP.NET 应用程序）是服务器——客户端的关系，应用程序使用数据库提供的服务完成数据存储、修改、查询等功能。应用程序使用数据库的第一步是连接到数据库，之后才能进行其他操作。

2.2.1　数据库连接类 DbConnection

.NET Framework 中的 System.Data.Common.DbConnection 类是一个抽象类，表示到数据库的连接。从 DbConnection 类派生出一组具体的数据库连接类，分别表示到一种特定数据源的连接，如图 2.2 所示。

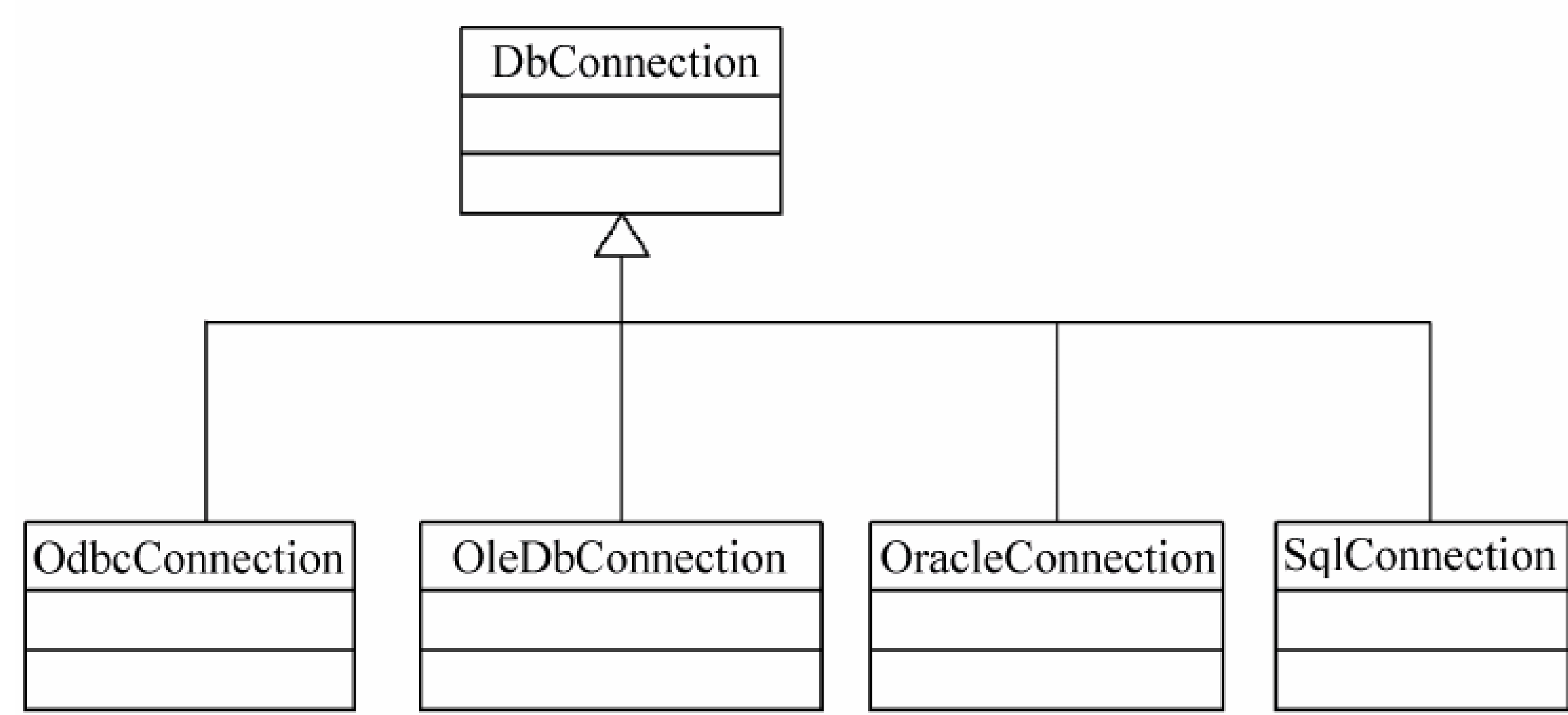


图 2.2　DbConnection 类及其派生类

图 2.1 中，OdbcConnection 类表示到 ODBC 数据源的连接，OleDbConnection 类表示到 OleDB 数据源的连接，OracleConnection 类表示到 Oracle 数据库的连接，SqlConnection 类表示到 MS SQL Server 数据库的连接。DbConnection 类的各个派生类的属性方法大致相同，下面以 SqlConnection 为例来说明连接到数据库的方法。

SqlConnection 类的主要属性见表 2-1。

表 2-1　SqlConnection类主要属性

属　　性	类　　型	读　　写	说　　明
ConnectionString	string	RW	用于连接到 SQL Server 数据库的连接字符串
Database	string	RW	连接到的数据库的名字
DataSource	string	RW	SQL Server 数据源名称
State	ConnectionState	R	连接的当前状态
ConnectionTimeout	int	RW	连接超时值（以秒为单位）

根据身份验证方式的不同，用于连接 SQL Server 的连接字符串也有所不同。如果使用

Windows 身份验证，则连接字符串具有如下形式。

```
Data Source= Initial Catalog=数据库名;Integrated Security=True
```

如果使用 SQL Server 身份验证，则连接字符串具有如下形式。

```
Server=SQLServer 实例名称;Database=数据库名; user id=用户名;password=密码
```

SqlConnection 类的 State 属性表示当前数据库连接状态，其值是一个 ConnectionState 枚举类型。ConnectionState 枚举类型共包含 6 个值，表示 6 种数据库连接状态。

- ❑ Closed：连接处于关闭状态。
- ❑ Connecting：正在与数据源连接。
- ❑ Open：连接处于打开状态。
- ❑ Executing：正在执行命令。
- ❑ Fetching：正在检索数据。
- ❑ Broken：连接中断。

SqlConnection 类的主要方法如下：

```
public SqlConnection();                                    //构造函数
public SqlConnection(string connectinString);              //构造函数
public override void Open();                               //打开连接
public override void Close();                              //关闭连接
public override void ChangeDatabase(string database);
                                                           //为打开的连接更改数据库
public SqlCommand CreateCommand();                         //创建一个 SQL 命令
```

2.2.2　连接到 SQL Server

使用 SqlConnection 连接到 SQL Server 数据库的方法就是先用连接字符串创建一个 SqlConnection 对象，然后调用 SqlConnection 类的 Connect()方法即可。连接字符串通常比较长，如果记不住，在 Visual Studio 中可以自动生成连接字符串，操作步骤如下。

（1）打开任意一个 ASP.NET 页面。

（2）从工具箱的“数据”选项卡中选择 SqlDataSource 控件拖放到页面上，单击控件的智能标记（控件右上角的小三角），则弹出智能标记面板，如图 2.3 所示。

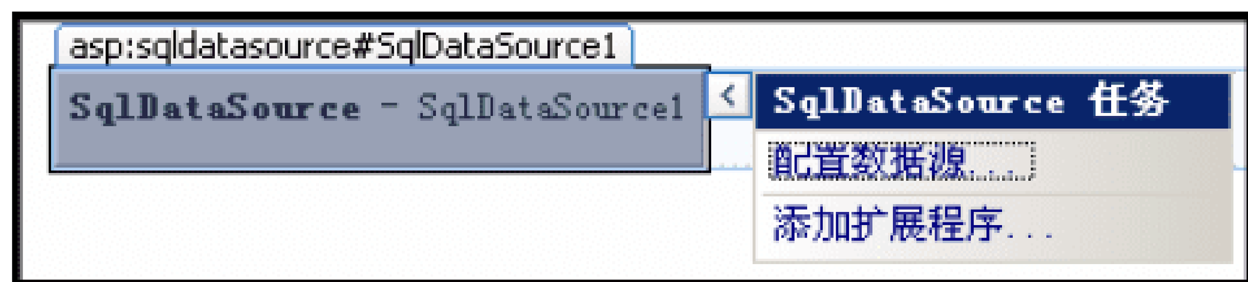


图 2.3　SqlDataSource 控件的智能标记面板

（3）在图 2.3 所示的智能标记面板中单击“配置数据源…”按钮，则弹出“配置数据源”向导对话框，如图 2.4 所示。

（4）单击“新建连接”按钮，则弹出“添加连接”对话框，如图 2.5 所示。

（5）在“添加连接”对话框中设置所要连接的数据库信息，然后单击“测试连接”按钮。提示连接成功后，单击“高级”按钮，就会弹出“高级属性”对话框，如图 2.6 所示。

在图 2.6 对话框的底部，就显示了当前数据库连接所对应的连接字符串。把这个字符串复制出来，就可以在程序中使用。

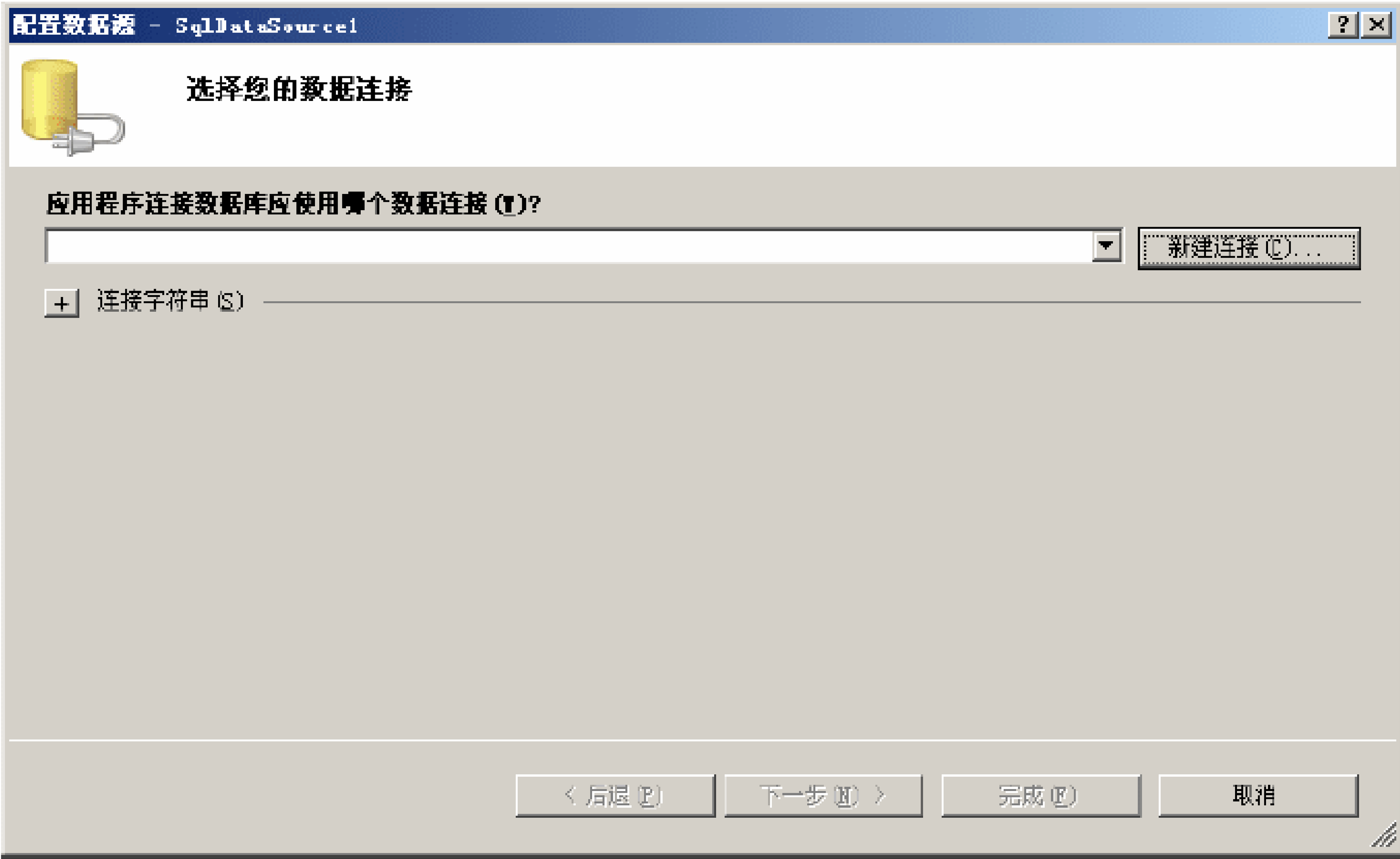


图 2.4 “配置数据源”对话框

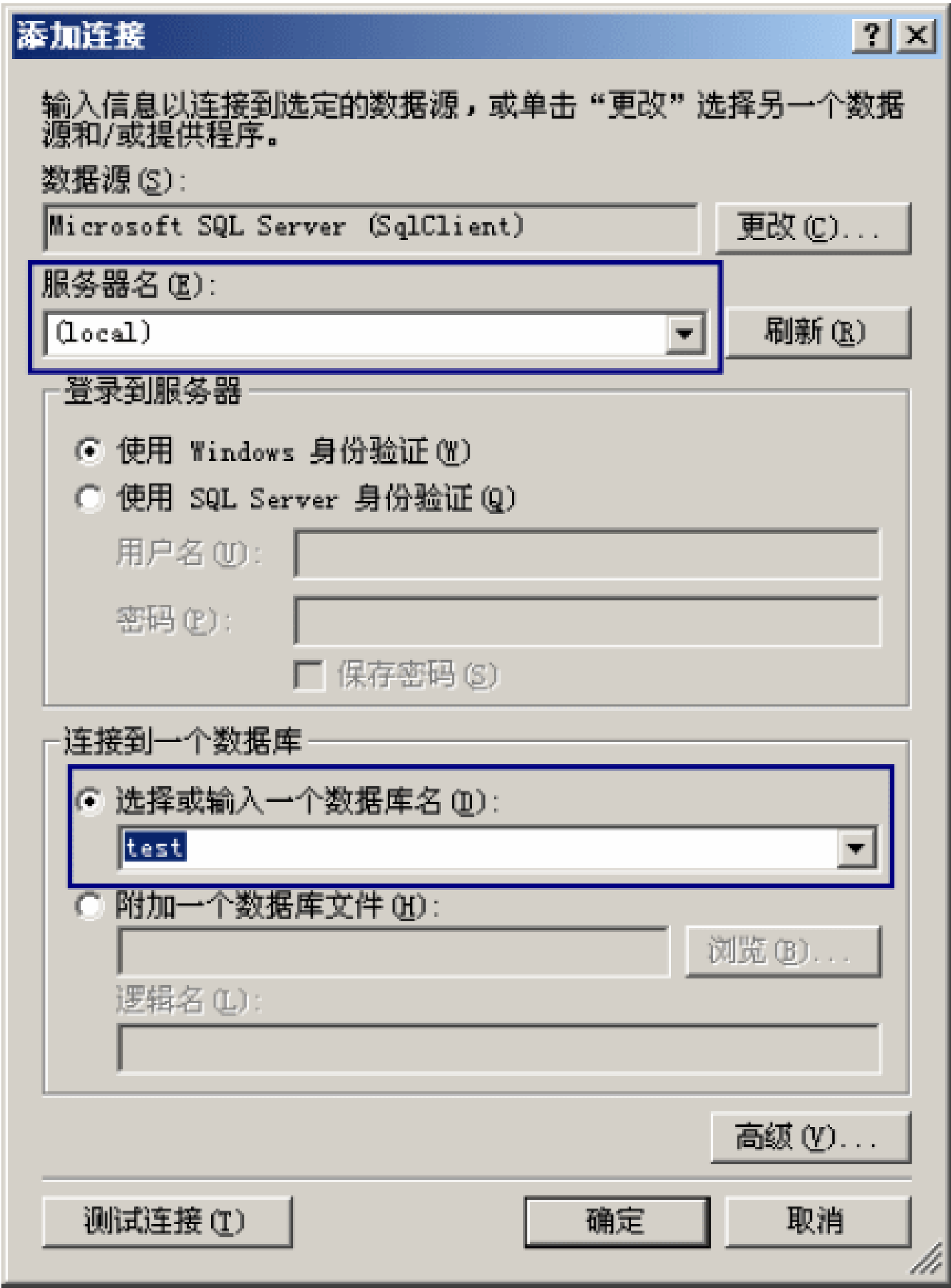


图 2.5 “添加连接”对话框

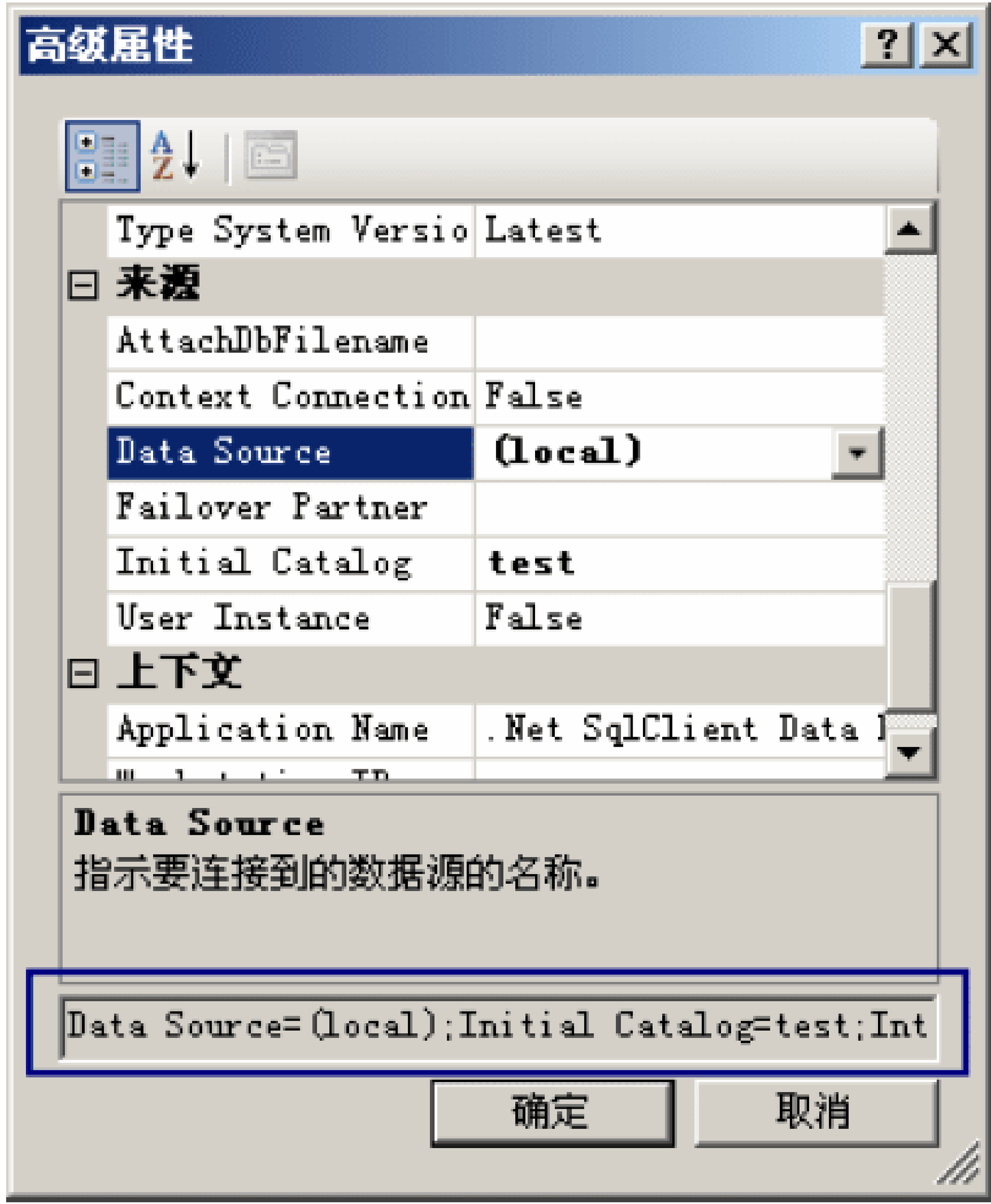


图 2.6 连接高级属性对话框

数据库连接是一种宝贵资源，在使用完毕连接以后，要及时调用 Close()方法关闭数据库连接。下面的代码演示了这一过程。

```
//连接字符串
string connString = "Data Source=(local);Initial Catalog=Northwind;
Integrated Security=True";
SqlConnection conn = new SqlConnection(connString);    //创建新的连接对象
conn.Open();                                           //打开连接
//在这里使用数据库连接
conn.Close();                                          //使用完毕后关闭连接
```

为了确保数据库连接能够总是被关闭，而不受代码中可能出现的异常影响，通常把关闭数据库连接的代码放到 finally 块中，代码如下：

```
string    connString="server=(local);database=Northwind;user    id=sa;
password=";
SqlConnection conn = new SqlConnection(connString);
try
{
    conn.Open();
    //在这里使用数据库连接
}
finally
{
    conn.Dispose ();
}
```

上述代码还有一种更简洁的书写方式，就是使用 using 语句，代码如下：

```
string connString = "server=(local);database=Northwind;user id=sa;password
=sa";
using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();
    //使用数据库连接 conn
}
```

使用 using 语句时，不论在 using 块中是否发生异常，在退出 using 块时都会调用对象的 Dispose()方法释放被使用的对象，在这里就是数据库连接。

注意：在 using 关键字后面括号里的对象必须实现 IDisposable 接口。

为了便于程序移植，通常把连接字符串写到配置文件 web.config 中，而不是硬编码到 C#代码中。这样，当需要连接的数据源发生变化时（例如把程序从开发环境部署到用户环境），就可以通过修改配置文件实现改变数据库连接，而不需要修改 C#代码和重新编译。把连接字符串保存到配置文件时，要保存在 ConnectionStrings 结点中，代码如下：

```
<connectionStrings>
    <add  name="database"  connectionString="Data  Source=.;  DataBase=
Northwind;  Integrated  Security  =True;  "   providerName="System.Data.
SqlClient" />
</connectionStrings>
```

由于配置文件中可以保存多个连接字符串，为了区别，每个连接字符串都有一个唯一的名字，在配置文件中用 name 属性标识。程序中可以根据名字找到某一个特定的连接字符串。在程序中获取连接字符串需要使用 ConfigurationManager 类。ConfigurationManager 类是专门用于访问配置文件的一个类，获取连接字符串的代码如下：

```
string connString = ConfigurationManager.ConnectionStrings["database"].
ConnectionString;
```

【例 2-1】 连接到 SQL Server。

本例演示如何用代码连接到 SQL Server 数据库。本例使用的数据库中 SQL Server 速成版（SQL Server Express）。如果要连接到其他版本，仅需要修改代码中的连接字符串即可。

提示： 使用 SQL Server 速成版的一个好处是可以直接在 Visual Studio 中添加数据库，从而数据库文件与项目中的其他文件在同一路径下，便于将来程序的部署。另外一个好处是程序运行不需要单独安装 SQL Server，.NET Framework 中自带了 SQL Server 速成版。如果用 SQL Server 标准版或者企业版，就需要单独安装 SQL Server 环境，还需要附加数据库等一些额外操作。

（1）创建一个 ASP.NET Web 应用程序 ADONET。

（2）在项目上右击，从弹出的快捷菜单中选择“添加新项”选项。在“添加新项”对话框中，选择“SQL Server 数据库”，单击“确定”按钮。这样就在项目中添加了一个 SQL Server 数据库。

（3）打开配置文件 web.config，在其中添加连接字符串，代码如下。

```
<connectionStrings>
    <add name="database" connectionString="Data Source=.\SQLEXPRESS;
    AttachDbFilename= |DataDirectory|\Database1.mdf;Integrated Security=
    True;User Instance=True"
         providerName="System.Data.SqlClient" />
</connectionStrings>
```

（4）在项目中添加一个页面 ConnectionSample.aspx，在页面上放置一个 Button 和一个 Label。代码如下：

```
<form id="form1" runat="server">
<div>
    <asp:Button ID="Button1" runat="server" Text="连接" onclick="Button1
    Click" /> <br />
    <asp:Label ID="Label1" runat="server" Text="输出信息"></asp:Label>
</div>
</form>
```

（5）为 Button 的 Click 事件编码，连接到数据库，然后关闭连接。代码如下：

```
protected void Button1 Click(object sender, EventArgs e)
{
    //连接字符串
    string connString = ConfigurationManager.ConnectionStrings["database"].
    ConnectionString;
    Label1.Text = "";
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
         //打开连接
        Label1.Text += "连接已经打开。<br/>";
        conn.Close();
         //关闭连接
        Label1.Text += "连接已经关闭。<br/>";
    }
}
```



图 2.7　连接数据库示例

（6）运行页面，单击“连接”按钮，运行结果如图 2.7 所示。

2.3　修 改 数 据

应用程序对数据库的操作总体可以分为两类：修改数据和读取数据。修改数据包含增加、删除、更新数据。本节将介绍 ADO.NET 中修改数据的相关类和方法。

2.3.1　数据库命令类 DbCommand

System.Data.Common.DbCommand 类表示要对数据库执行的 SQL 语句或存储过程。DbCommand 类是一个抽象类，.NET Framework 为不同的数据库管理系统分别创建了从 DBCommand 类派生的具体的数据库命令类，如图 2.8 所示。

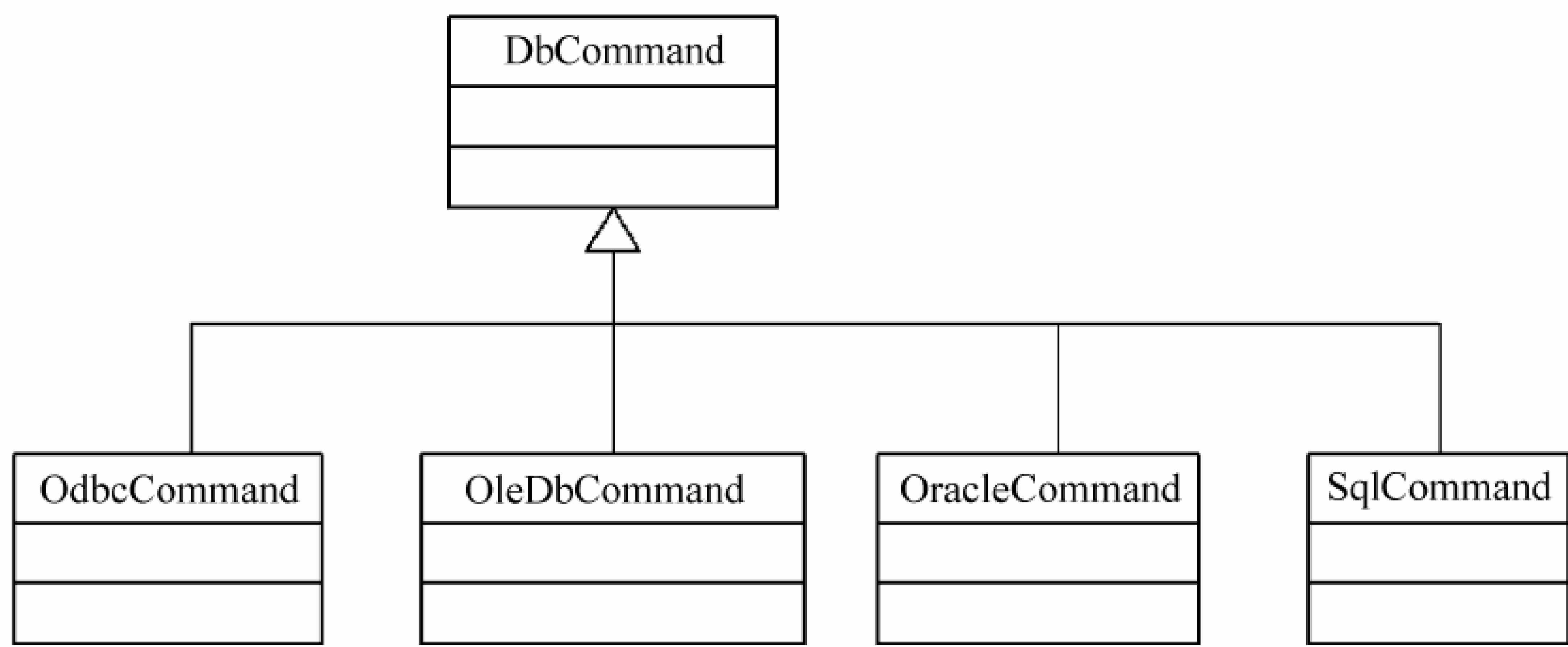


图 2.8　DbCommand 类及其派生类

图 2.7 中，OdbcCommand 类表示 ODBC 数据源的命令，OleDbConnection 类表示 OleDB 数据源的命令，OracleConnection 类表示 Oracle 数据库的命令，SqlConnection 类表示 MS SQL Server 数据库的命令。这一组数据库命令类的使用大致相似，下面以 SqlCommand 为例介绍。

SqlCommand 类的主要属性见表 2-2。

表 2-2　SqlCommand类主要属性

属　　性	类　　型	读　　写	说　　明
CommandType	CommandType	RW	命令类型，如存储过程、SQL 语句等
CommandText	string	RW	命令文本
Connection	SqlConnection	RW	此命令所使用的 SqlConnection 对象
Parameters	SqlParameterCollection	R	命令的参数
CommandTimeout	int	RW	以秒为单位的命令超时时间
Transaction	SqlTransaction	RW	命令所属的事务对象

SqlCommand 类的 CommandText 属性是一个字符串，表示命令的文本，其值可以是一组 SQL 语句，如 select * from Customers，也可以是一个存储过程，如 SalesByCategory。

SqlCommand 类的 CommandType 属性是一个 CommandType 枚举类型的值，表示 SqlCommand 对象的 CommandText 是什么含义。CommandType 枚举包含以下值。

- ❑ Text：CommandText 中包含 SQL 命令文本。
- ❑ StoredProcedure：CommandText 中包含存储过程名称。
- ❑ TableDirect：CommandText 中包含数据库表的名称。

Parameters 和 Transaction 属性将在本章后续内容中详细介绍。

SqlCommand 类主要方法如下。

（1）构造函数

SqlCommand 类共有 4 个构造函数，代码如下：

```
public SqlCommand();//默认构造函数
public SqlCommand(string commandText); //用指定的 SQL 命令文本初始化一个命令
public SqlCommand(string commandText, SqlConnection connection);
//用指定的 SQL 命令文本和数据库连接对象初始化一个命令
//用指定的 SQL 命令文本、数据库连接和事务对象初始化一个命令
public    SqlCommand(string    commandText,    SqlConnection    connection,
SqlTransaction transaction);
```

（2）执行命令

SqlCommand 类可以以几种不同的方式执行，代码如下：

```
public override int ExecuteNonQuery();      //执行 SQL 命令并返回受影响的行数
public override object ExecuteScalar();     //执行 SQL 命令，并返回结果集中第一
                                            //行第一列的值
public SqlDataReader ExecuteReader();       //执行 SQL 命令，并返回一个 SqlData-
                                            //Reader 对象
public SqlDataReader ExecuteReader(CommandBehavior behavior);
```

如果要执行的 SQL 命令没有返回结果，如 INSERT、DELETE、CREATE TABLE 等命令，那么应该调用 ExecuteNonQuery()方法。如果要执行的命令仅返回一个值，如 SELECT COUNT(*)命令等，那么应该调用 ExecuteScalar()方法。如果要执行的命令会返回多行多列，则应该使用 ExecuteReader()方法。ExecuteReader()方法返回一个 SqlDataReader 对象，关于 SqlDataReader，将在后续内容中介绍。

2.3.2　命令参数 DbParameter

执行 SQL 命令时，大多数情况下都需要向 SQL 语句或者存储过程传递参数。在.NET Framework 中，使用 System.Data.Common.DbParameter 类表示 SQL 命令的参数。DbParameter 是一个抽象类。如同 DbConnection 和 DbCommand 类一样，对于不同的数据库管理系统，.NET Framework 中有不同的 DbParameter 类的派生类，包括 OdbcParameter、OleDbParameter、OracleParameter 和 SqlParameter。SqlParameter 类的主要属性见表 2-3。

表 2-3　SqlParameter类主要属性

属　　性	类　　型	读　　写	说　　明
ParameterName	string	RW	参数名称
DbType	SqlDbType	RW	参数的 SQL 数据类型
Direction	ParameterDirection	RW	参数传递方向（输入输出）
IsNullable	bool	RW	参数是否可以为空
Value	object	RW	参数的值

SqlParameter 类的主要构造函数如下：

```
public SqlParameter(); //默认构造函数
public SqlParameter(string paramName, object value);
                    //用参数名称和值初始化一个新的 SqlParameter 对象
public SqlParameter(string paramName, SqlDbType dbType);
                    //创建一个具有指定名称和类型的 SqlParameter 对象
public SqlParameter(string paramName, SqlDbType dbType, int size);
                    //创建一个具有指定名称、类型、大小的 SqlParameter 对象
```

要执行带参数的 SQL 语句或者存储过程时，必须向 SqlCommand 中添加所需的参数。方法是创建 SqlParameter 类的实例，并调用 SqlCommand.Parameters.Add()方法将参数添加到命令中。

2.3.3　修改数据

对数据库数据的修改主要包括插入（INSERT）、删除（DELETE）、更新（UPDATE）3 种操作，这 3 种操作通常分别对应于一条 SQL 语句。如前所述，ADO.NET 的 DbCommand 类表示 SQL 命令，可以用 DbCommand 执行各种数据修改。

用 ADO.NET 执行一条 SQL 命令通常包含以下几个步骤。

（1）获取连接字符串，创建到数据库的连接。

（2）打开连接。

（3）创建命令对象。

（4）为命令添加参数。

（5）执行命令，获得命令结果。

（6）关闭命令。

（7）关闭连接。

（8）处理命令结果。

【例 2-2】 用户注册。

本例以用户注册页面为例，演示如何用 ADO.NET 向数据库插入数据。

（1）创建一个 ASP.NET 应用程序。

（2）在项目中添加一个数据库，在数据库中添加一个表 LoginUser，为表添加三列，各列的名称、字段类型如下：

- ❑ UserID：登录用户 ID，nvarchar(10)类型，不可为空，主键。
- ❑ UserName：用户名称，nvarchar(20)类型，不可为空。
- ❑ Password：登录密码，nvarchar(20)类型，不可为空。

（3）在项目中添加一个注册页面 RegisterPage.aspx，在页面中放置 3 个 TextBox，用于输入注册信息，放置一个 Button，以完成注册功能，页面代码如下：

```
<form id="form1" runat="server">
   <div>
   用户登录 ID：<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
   <br />
   用户名称：<asp:TextBox ID="TextBox2" runat="server"></asp:TextBox><br />
   登录密码：<asp:TextBox ID="TextBox3" runat="server" TextMode="Password">
```

```
</asp:TextBox> <br />
    <asp:Button ID="Button1" runat="server" Text="注册" /><br />
    </div>
</form>
```

（4）在“注册”按钮的 Click 事件中编写代码，完成用户注册功能。代码如下：

```
protected void Button1 Click(object sender, EventArgs e)
{
    //从配置文件中得到连接字符串
    string   s   =   ConfigurationManager.ConnectionStrings["database"].
    ConnectionString;
    SqlConnection connection = new SqlConnection(s);       //创建连接对象
                                                //构建插入 SQL 语句
    string sql = "insert into loginuser (userid,username,password)"
    +" values (@id, @name, @pass)";
    SqlCommand command = new SqlCommand(sql,connection);  //创建命令对象
    SqlParameter p1 = new SqlParameter("@id",TextBox1.Text);
                                                //创建一个命令参数 userid
    command.Parameters.Add(p1);                 //把命令参数附加到命令
    //构建和附加其他两个参数
    SqlParameter p2 = new SqlParameter("@name", TextBox2.Text);
    command.Parameters.Add(p2);
    SqlParameter p3 = new SqlParameter("@pass", TextBox3.Text);
    command.Parameters.Add(p3);
    string message="";
    try
    {
        connection.Open();                      //打开连接
        int n = command.ExecuteNonQuery();      //执行命令，得到受影响的行数
        if (n == 1)
            message = "注册成功。用户信息已经保存在数据库。";
        else
            message = "数据未能保存。";
    }
    catch (Exception ex)
    {
        //要注意处理错误提示信息中的换行符
        message = "保存用户信息过程中出错。"
            + ex.Message.Replace("\r", "\\r ").Replace("\n","\\n");
    }
    finally
    {
        command.Dispose();
        connection.Close();
    }
    //向客户端返回 JavaScript，提示执行结果
    Page.ClientScript.RegisterStartupScript(this.GetType(),
        "register", "<script>alert(\""+message+"\");</script>");
}
```

（5）运行 RegisterPage.aspx 页面，测试注册功能。运行界面如图 2.9 所示。

【例 2-3】 修改密码。

本例以修改用户密码页面为例，说明如何用 ADO.NET 更新数据库数据。

（1）打开例 2-2 所创建的 ASP.NET 项目。

（2）在项目中添加一个页面 ChangePass.aspx，在页面上放置相应控件以进行密码修改，

页面代码如下：

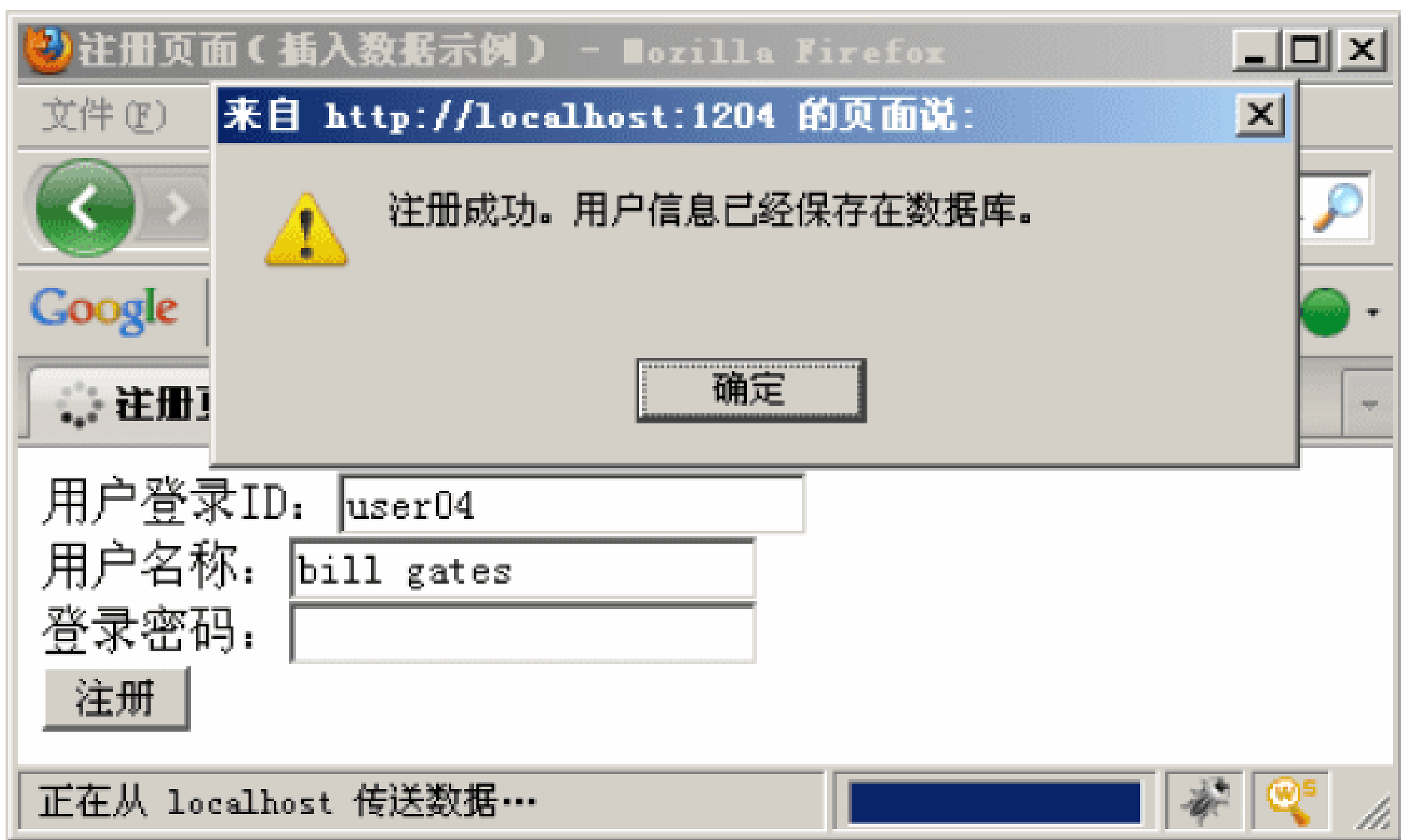


图 2.9　用户注册页面

```
<form id="form1" runat="server">
<div>
用户 ID：<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox><br />
原密码：
<asp:TextBox  ID="TextBox2"  runat="server"  TextMode="Password"  ></asp:
TextBox><br />
新密码：
<asp:TextBox ID="TextBox3" runat="server" TextMode="Password"></asp:Text
Box><br />
<asp:Button ID="Button1" runat="server" Text="修改密码" onclick="Button1
Click" /><br />
</div>
</form>
```

（3）为 ChnagePass.aspx 页面的“修改密码”按钮编写代码，完成密码修改功能。代码如下：

```
protected void Button1_Click(object sender, EventArgs e)
{
    //从配置文件中得到连接字符串
    string s = ConfigurationManager.ConnectionStrings["database"].
    ConnectionString;
    SqlConnection connection = new SqlConnection(s);    //创建连接对象
    //构建更新用的 SQL 语句
    string sql = "update loginuser set password = @newpass "
        +"where userid = @id and password = @oldpass";
    SqlCommand command = new SqlCommand(sql, connection);  //创建命令对象
    SqlParameter p1 = new SqlParameter("@id", TextBox1.Text);
                                            //创建一个命令参数 userid
    command.Parameters.Add(p1);             //把命令参数附加到命令
    //构建和附加其他两个参数
    SqlParameter p2 = new SqlParameter("@oldpass", TextBox2.Text);
    command.Parameters.Add(p2);
    SqlParameter p3 = new SqlParameter("@newpass", TextBox3.Text);
    command.Parameters.Add(p3);
    string message = "";
    try
    {
        connection.Open();                  //打开连接
        int n = command.ExecuteNonQuery();  //执行命令，得到受影响的行数
        if (n == 1)
```

```
            message = "密码修改成功。";
        else
            message = "不存在此用户或者原始密码不对。";
    }
    catch (Exception ex)
    {
        //要注意处理错误提示信息中的换行符
        message = "更改密码过程中出错。"
            + ex.Message.Replace("\r", "\\r ").Replace("\n", "\\n");
    }
    finally
    {
        command.Dispose();
        connection.Close();
    }
    //向客户端返回 JavaScript，提示执行结果
    Page.ClientScript.RegisterStartupScript(this.GetType(),
        "register", "<script>alert(\"" + message + "\");</script>");
}
```

（4）运行 ChangePass.aspx 页面，运行结果如图 2.10 所示。

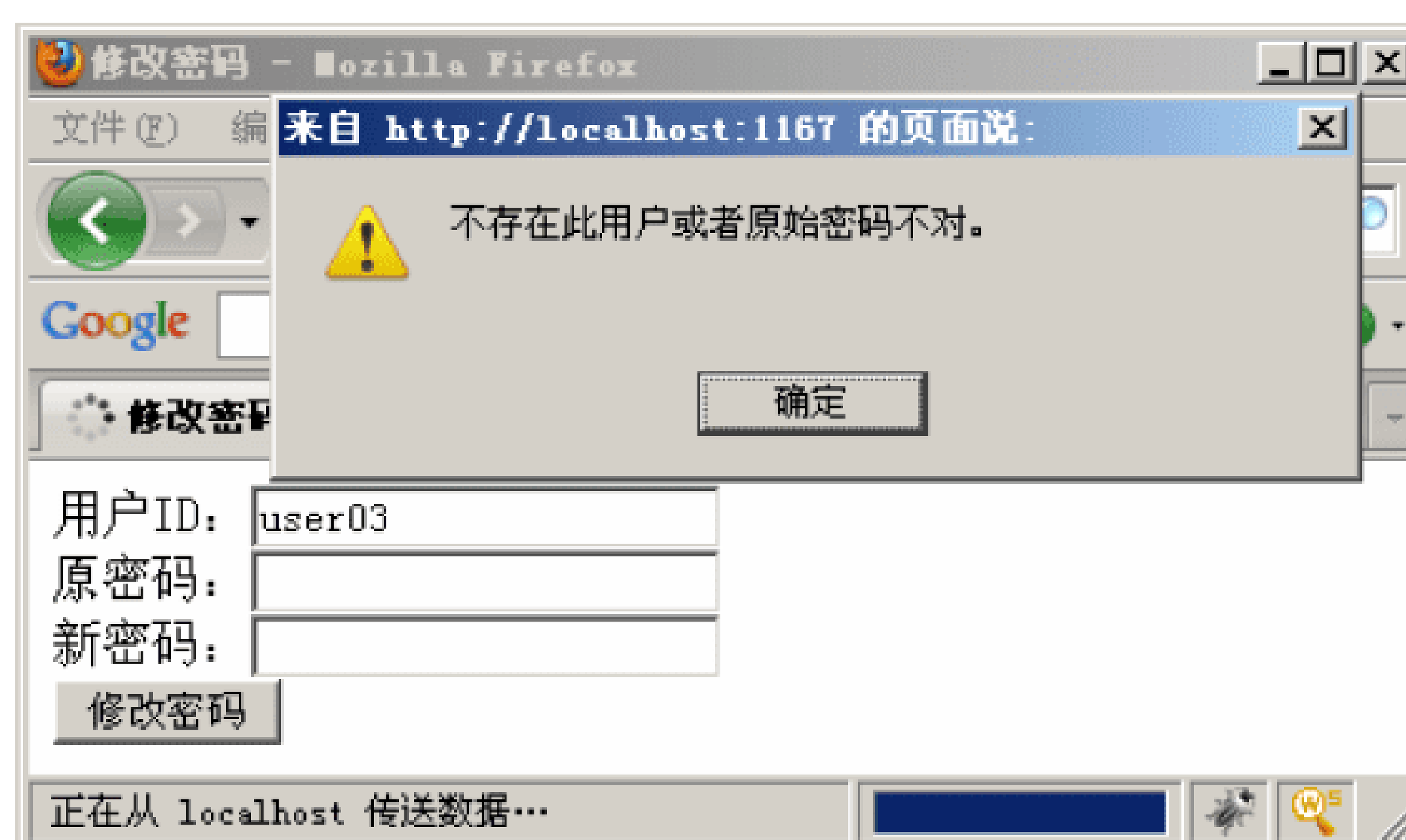


图 2.10　修改密码页面

2.4　查 询 数 据

由于 SQL 查询操作的复杂性，使用 ADO.NET 从数据库查询数据要比修改数据更加复杂。对数据库的查询要用到 ADO.NET 中的许多类，除了在修改数据时用到的 DbCommand、DbConnection、DbParameter 类以外，还要到 DataReader、DataSet、DataAdapter 等类。

2.4.1　查询单个值

对数据库进行查询时，大多数查询可能返回多个数据，例如查询满足一定条件的学生信息（姓名、学号、班级等），而有的查询只会返回一个数据，例如，查询满足条件的学生数量（会返回一个整数）。这两种不同的查询方式，在 ADO.NET 中用不同的语句来完成，返回单个值的查询语句用 DbCommand.ExecuteScalar 语法执行，返回多个值的查询语句操作较为复杂，会在本章后续内容中加以介绍。

返回单个值的查询语句通常都是聚合函数，如 COUNT、MAX、AVERAGE 等，但某些时候非聚合函数也会返回单个值，例如以主键为条件进行查询单个列时。以 SQL Server 提供的 Northwind 示例数据库为例，下列 SQL 查询语句都是返回单个值的查询语句。

```
select count(*) from customers where Country='UK'
/*查询来自 UK 的顾客总数*/
select sum(UnitPrice*Quantity) from [Order Details] where OrderID='10248'
/*查询订单的总金额*/
select Phone from Customers where CustomerID='ALFKI'
/*查询 ALFKI 顾客的联系电话*/
```

【例 2-4】 用户信息统计。

本例将统计注册用户汇总信息，注册用户一共多少人，男女各多少人。

（1）打开例 2-2 所创建的 ASP.NET 项目。

（2）打开项目中的数据库，为 LoginUser 表添加两个字段。

- ❑ RegisterDate：用户注册日期，datetime 类型，可空。
- ❑ Sex：性别，nchar(1)类型，可空，M 为男，F 为女。

（3）在项目中添加一个页面 UserStatistics.aspx，在页面上放置相应控件以统计用户数并显示结果，页面代码如下：

```
<form id="form1" runat="server">
<div>
<asp:Button ID="Button1" runat="server" Text="统计用户数" onclick=
"Button1_Click" /><br />
总用户数:
<asp:Label ID="total" runat="server" Text="Label"></asp:Label><br />
其中男用户
<asp:Label ID="male" runat="server" Text="Label"></asp:Label>人，
女用户<asp:Label ID="female" runat="server" Text="Label"></asp:Label>人，
未填写性别者<asp:Label ID="unknown" runat="server" Text="Label"></asp:
Label>人。
</div>
</form>
```

（4）为“统计用户数”按钮编写代码，执行查询并显示结果。代码如下：

```
protected void Button1_Click(object sender, EventArgs e)
{
    //从配置文件中得到连接字符串
    string s = ConfigurationManager.ConnectionStrings["database"].
    ConnectionString;
    SqlConnection connection = new SqlConnection(s);        //创建连接对象
    string sql = "select count (*) from LoginUser";
                                                 //构建 SQL 语句（查询总用户数）
    SqlCommand command = new SqlCommand(sql, connection);  //创建命令对象
    try
    {
        connection.Open();                       //打开连接
        int n = Convert.ToInt32(command.ExecuteScalar());
                                                 //执行命令，得到查询结果
        total.Text = n.ToString();
        //修改 SQL 语句，查询性别为空的用户
        command.CommandText = "select count(*) from LoginUser where Sex is
        null";
```

```
        n = Convert.ToInt32(command.ExecuteScalar()); //执行查询，并显示结果
        unknown.Text = n.ToString();
        //修改 SQL 语句，根据性别为 M 或 F 进行查询
        command.CommandText = "select count(*) from LoginUser where Sex=
        @sex";
        //为查询参数赋值为 M，查询男用户数
        SqlParameter p1 = new SqlParameter("@sex", 'M');
        command.Parameters.Add(p1);
        n = Convert.ToInt32(command.ExecuteScalar());
        male.Text = n.ToString();
        //为查询参数赋值为 F，查询男用户数
        command.Parameters[0].Value = "F";
        n = Convert.ToInt32(command.ExecuteScalar());
        female.Text = n.ToString();
    }
    catch (Exception ex)
    {
        string message = "执行命令过程中出错。"
            + ex.Message.Replace("\r", "\\r ").Replace("\n", "\\n");
        Response.Write(message);
    }
    finally
    {
        command.Dispose();
        connection.Close();
    }
}
```

（5）运行 UserStatistics.aspx 页面，运行结果如图 2.11 所示。

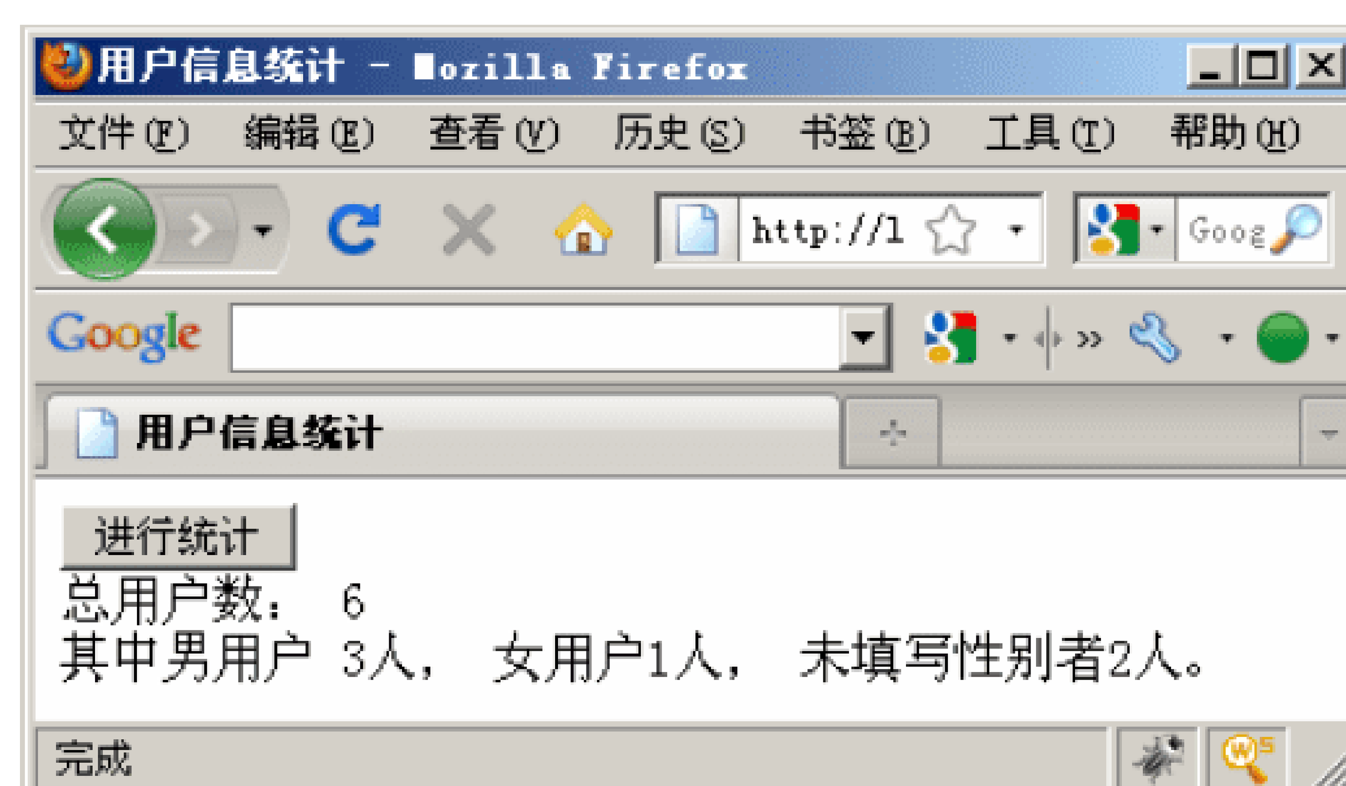


图 2.11　用户统计页面

2.4.2　数据读取器 DataReader

DataReader 以只读、单向的方式访问数据库。只读，指的是使用 DataReader 只可以从数据库中检索数据，而不可以对数据库的数据进行修改；单向，指的是用 DataReader 读取数据时，只能按一定的顺序逐行读取数据，不能回头读取已经被读取过的数据。

在.NET Framework 中，实现 DataReader 的基类是 System.Data.Common.DbDataReader。针对不同的数据库系统，从 DbDataReader 类派生的类有 ObdeDataReader、OleDbDataReader、OracleDataReader SqlDataReader 等。下面以 SqlDataReader 为例说明 DataReader 的使用。

SqlDataReader 类的主要属性见表 2-4。

表 2-4　SqlDataReader类主要属性

属　性	类　型	读　写	说　明
FieldCount	int	R	当前行中的列数
HasRows	bool	R	DataReader 中是否包含行
IsClosed	bool	R	DataReader 是否已经关闭

SqlDataReader 类声明了两个索引器，可以用 int 类型的列序号或者 string 类型的列名为索引。两个索引器的定义如下：

```
public override object this [int columnOrdinal] {get;}
//根据列序号取得 DataReader 中某列的值
public override object this [string columnName] {get;}
//根据列序号取得 DataReader 中某列的值
```

提示：若要得到一个 SqlDataReader 对象，必须调用 SqlCommand 对象的 ExecuteReader()方法，而不能调用 SqlDataReader 的构造函数。SqlDataReader 类没有公共构造函数。

SqlDataReader 类的主要方法如下：

（1）操作 SqlDataReader。

```
public void Close()                         //关闭 SqlDataReader
public virtual bool Read();                 //使 SqlDataReader 前进到下一条记录
public virtual bool NextResult();           //使 SqlDataReader 前进到下一个结果
```

（2）获取列信息。

```
public virtual Type GetFieldType(int i);        //得到指定列的数据类型
public virtual string GetName(int i);           //得到指定列的列名
public virtual int GetOrdinal(string name);     //得到具有指定名称的列的序号
public override bool IsDBNull(int i);           //判断指定列是否为空
```

（3）获取列的值。

```
public override object GetValue(int i);         //得到第 i 列的值
//以下方法以特定数据类型返回指定列的值
public virtual bool GetBoolean(int i);
public virtual byte GetByte(int i);
public virtual byte GetByte(int i);
public virtual char GetChar(int i);
public virtual DateTime GetDateTime(int i);
public virtual decimal GetDecimal(int i);
public virtual double GetDouble(int i);
public virtual float GetFloat(int i);
public virtual short GetInt16(int i);
public virtual int GetInt32(int i);
public virtual long GetInt64(int i);
public virtual string GetString(int i);
```

SqlDataReader 每一次读取只从数据库读取一行数据，因此，如果要处理多行数据，则在处理多行期间，数据库连接必须始终保持可用。在使用 SqlDataReader 时，与此 SqlDataReader 相关联的 SqlConnection 正忙于为 SqlDataReader 服务，而无法执行任何其他操作。

从 SqlDataReader 中得到某列数值有 4 种方法，代码如下：

```
//设 reader 为 SqlDataReader 类的实例
string id1 = reader.GetString (0);
                                    //方法一：使用强类型的 Get 方法得到列的值
string id2 = (string)reader.GetValue(0);
                                    //方法二：使用弱类型的 GetValue 得到列的值
string id3 = (string)reader[0];     //方法三：使用索引器（列序号作索引）
string id4 = (string)reader["CustomerID"];
                                    //方法四：使用索引器（列名作索引）
```

【例 2-5】 读取数据。

本例演示如何通过 DataReader 从数据库读取数据。

（1）打开例 2-4 所创建的项目。

（2）在项目中添加一个页面 DataReaderPage1.aspx。

（3）在页面上放置一个 ASP.NET 服务器端 Table，用于显示数据，放置一个 Button，用于查询数据。页面代码如下：

注意： 在 ASP.NET 页面上显示数据一般并不使用 Table，而是使用专门的数据绑定控件，如 GridView、DataList 等。由于本书在后续章节中才会讲解数据绑定控件，为了不使用后面没有讲到的内容，本章显示数据时使用 Table。

```
<form id="form1" runat="server">
<div>
   <asp:Button ID="Button1" runat="server" Text="查询" onclick="Button1_
   Click" />
   <h3>查询结果</h3>
   <asp:Table ID="result" runat="server" BorderStyle="Solid" Border
   Width="1px"
      GridLines="Both" >
   </asp:Table>
</div>
</form>
```

（4）在按钮的 Click 事件中读取数据，并显示在页面上。代码如下：

```
protected void Button1_Click(object sender, EventArgs e)
{
   //从配置文件中得到连接字符串
   string s = ConfigurationManager.ConnectionStrings["database"].
   ConnectionString;
   //创建和使用连接对象
   using (SqlConnection connection = new SqlConnection(s))
   {
      string sql = "select UserID,UserName,Password,Sex from LoginUser";
      SqlCommand command = new SqlCommand(sql, connection);
      connection.Open();                              //打开连接
      //得到 DataReader 对象
      using (SqlDataReader reader = command.ExecuteReader())
      {
         while (reader.Read())                        //循环读取数据
         {
            TableRow row = new TableRow();            //在页面表上添加一行
            result.Rows.Add(row);
            //将 4 个字段添加到行中
```

```
                for (int i = 0; i < 3; i++)
                {
                    TableCell cell = new TableCell();
                    cell.Text = Convert.ToString(reader[i]);
                    row.Cells.Add(cell);
                }
            }
        }
        command.Dispose();
        connection.Close();
    }
}
```

（5）运行页面，运行结果如图 2.12 所示。



图 2.12　数据读取器示例

2.5　数据集和数据适配器

.NET Framework 中的数据集 DataSet 是一个内存中的数据库模型，数据库中的各种元素如表、行、列、外键关系等都可以再映射到 DataSet，因此使用 DataSet 可以准确反映数据库。数据适配器 DataAdapter 的作用相当于数据库和 DataSet 之间的一个桥梁，一方面可以把数据从数据库读取出来放在 DataSet 中，另一方面还可以把 DataSet 中被修改的数据写回数据库。本节将介绍 DataSet 和 DataAdapter 的使用。

2.5.1　数据集 DataSet 概述

.NET Framework 中的 DataSet 类就是一个数据库的模型，数据库中包含的许多元素在 DataSet 中都有对应的类。DataSet 中可以有数据表和视图，表之间有关系，表中有行和列，表中每一列都有一种特定的数据类型，可以或者不可以为空，可以被设置为主键等等，所有这些属性，都使 DataSet 看起来像是一个数据库。DataSet 对象模型如图 2.13 所示。

数据表 DataTable、数据行 DataRow 和数据列 DataColumn 是 DataSet 体系中最基本的组成元素。DataTable 表示 DataSet 中一个数据表，DataColumn 表示 DataTable 中的一列，DataRow 表示 DataTable 中的一行。DataTable、DataColumn、DataRow 类之间的关系与数

据库中表、列、行之间的关系相同。

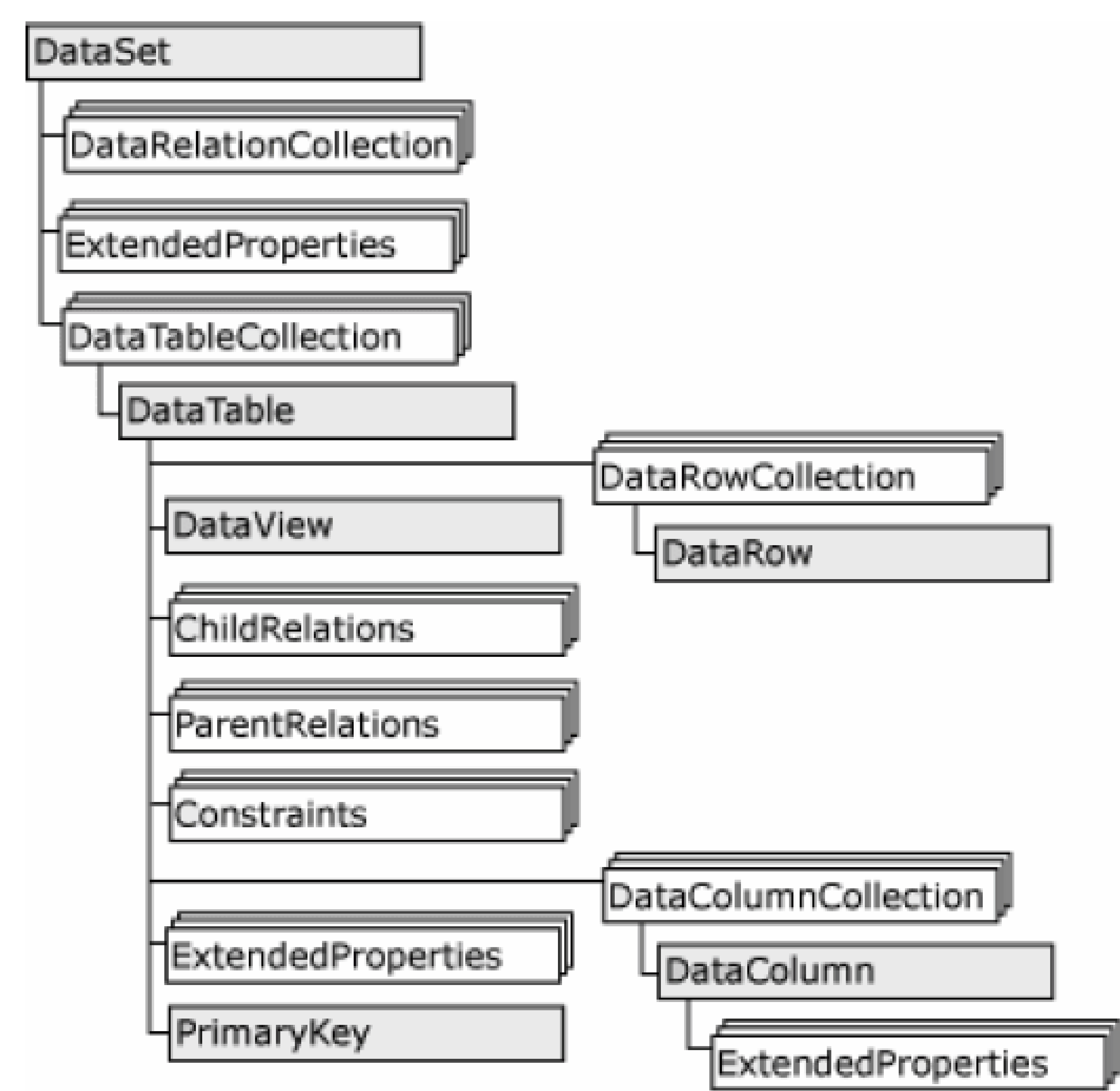


图 2.13　DataSet 对象模型

2.5.2　数据适配器 DataAdapter 概述

DataAdapter 的作用相当于数据库和 DataSet 之间的一个桥梁，一方面可以把数据从数据库读取出来放在 DataSet 中，另一方面还可以把 DataSet 中被修改的数据写回数据库。这个功能是通过 SqlDataAdapter 类的 4 个 SqlCommand 属性实现的，代码如下：

```
//SelectCommand 用于从数据库读取数据到 DataSet
public SqlCommand SelectCommand{get; set;}
//InsertCommand 用于把 DataSet 中的新增行插入到数据库中
public SqlCommand InsertCommand{get; set;}
//DeleteCommand 用于把 DataSet 中被删除的行从数据库中删除
public SqlCommand DeleteCommand{get; set;}
//UpdateCommand 用于把 DataSet 中数据的修改更新到数据库中
public SqlCommand UpdateCommand{get; set;}
```

SqlDataAdapter 的构造函数如下：

```
//默认构造函数
public SqlDataAdapter();
//创建 SqlDataAdapter 新实例，并用指定的 SqlCommand 作为其 SelectCommand
public SqlDataAdapter(SqlCommand selectCommand);
//创建 SqlDataAdapter 新实例，并指定其 SelectCommand 的连接和命令文本
public  SqlDataAdapter(string  selectCommandText,  SqlConnection  select
Connection);
//创建 SqlDataAdapter 新实例，并指定其 SelectCommand 的连接字符串和命令文本
public SqlDataAdapter(string selectCommandText, string selectConnection
String);
```

2.5.3　填充数据

使用 SqlDataAdapter 类的 Fill()方法可以从数据库中读取数据并填充到 DataSet 中。使

用 SqlDataAdapter.Fill()方法填充数据时，将执行 SqlDataAdapter 的 SelectCommand，并将检索到的数据填充到 DataSet 的一个 DataTable 中。

SqlDataAdapter 类的 Fill()方法有多个重载版本，常用的有如下 3 个版本。

```
public override int Fill(DataSet dataset);
                                    //填充 DataSet 中名为 Table 的 DataTable
pubic override int Fill(DataTable table);  //填充指定的 DataTable
pubic override int Fill(DataSet dataset, string tableName);
                                            //填充具有指定名称的 DataTable
```

在数据填充时，如果 DataSet 中不存在指定名称的表，则创建一个具有指定名称的表，并把数据填充其中。如果 DataSet 中已经存在相同名称的 DataTable，则可分为两种情况，一种情况是 DataTable 有主键，那么将根据主键匹配刷新相应的数据；另一种情况是 DataTable 没有主键，则在 DataTable 中原有行的后面追加新检索到的行。

注意：DataSet 中 DataTable 的名称与数据库中的表名没有必然联系，二者可以相同也可以不同。当使用 SqlDataAdapter 填充 DataTable 时，如果没有指定即将填充的表名，则表名默认为 Table，并不会自动使用 SELECT 语句中的表名。

在调用 SqlDataAdapter.Fill()方法以前，并不要求所使用的连接已经打开，但要求连接必须是合法的。如果在 Fill()方法以前连接是关闭的，那么 Fill()方法将打开连接，读取和填充数据，然后关闭连接。如果在 Fill()方法以前连接是打开的，那么 Fill()方法后连接仍然打开。

用 DataAdapter.Fill()方法填充 DataSet 的整个过程，如图 2.14 所示。

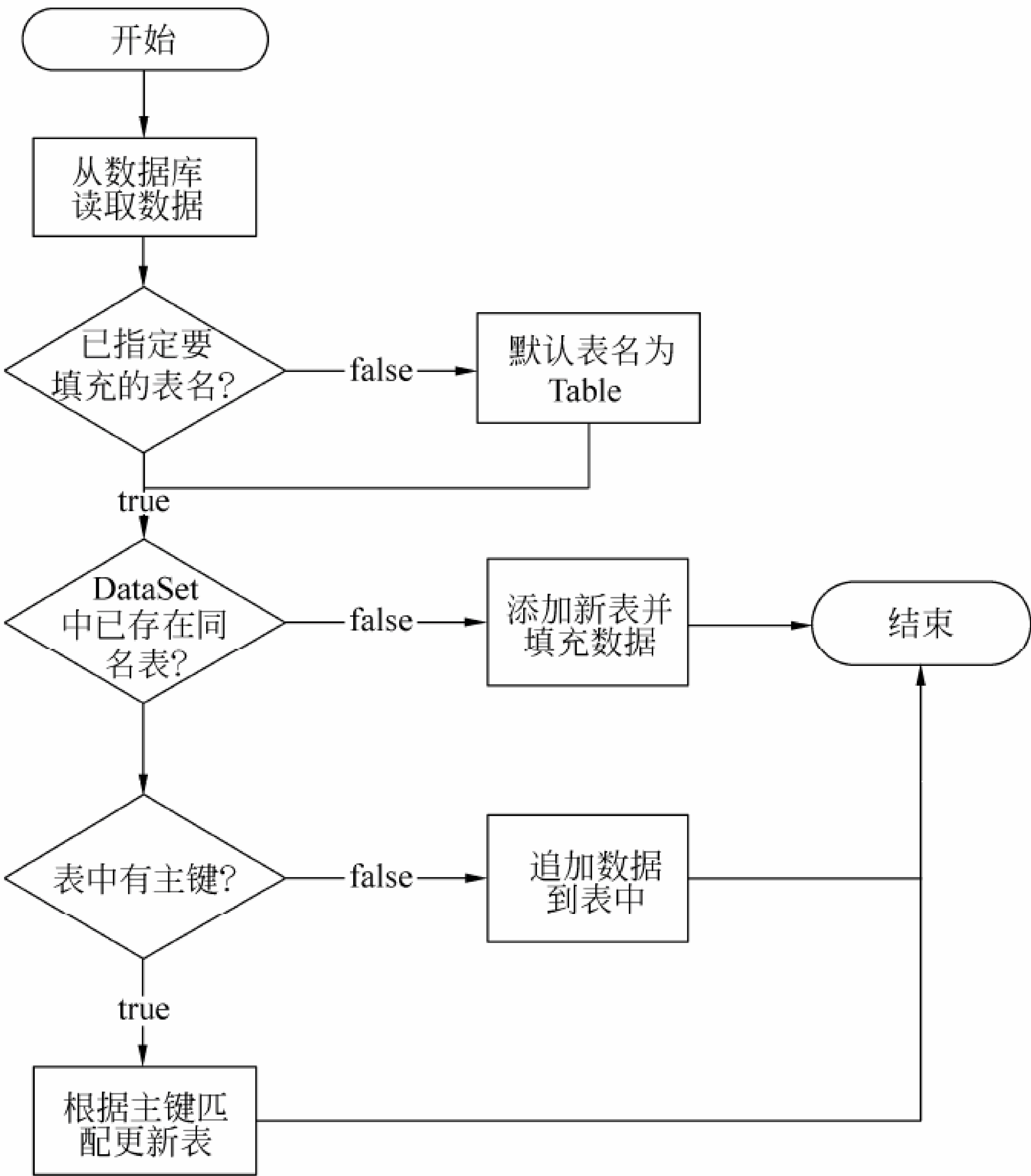


图 2.14　填充 DataSet 过程

【例 2-6】 用 DataAdatper 填充数据。

（1）打开例 2-4 所创建的项目。

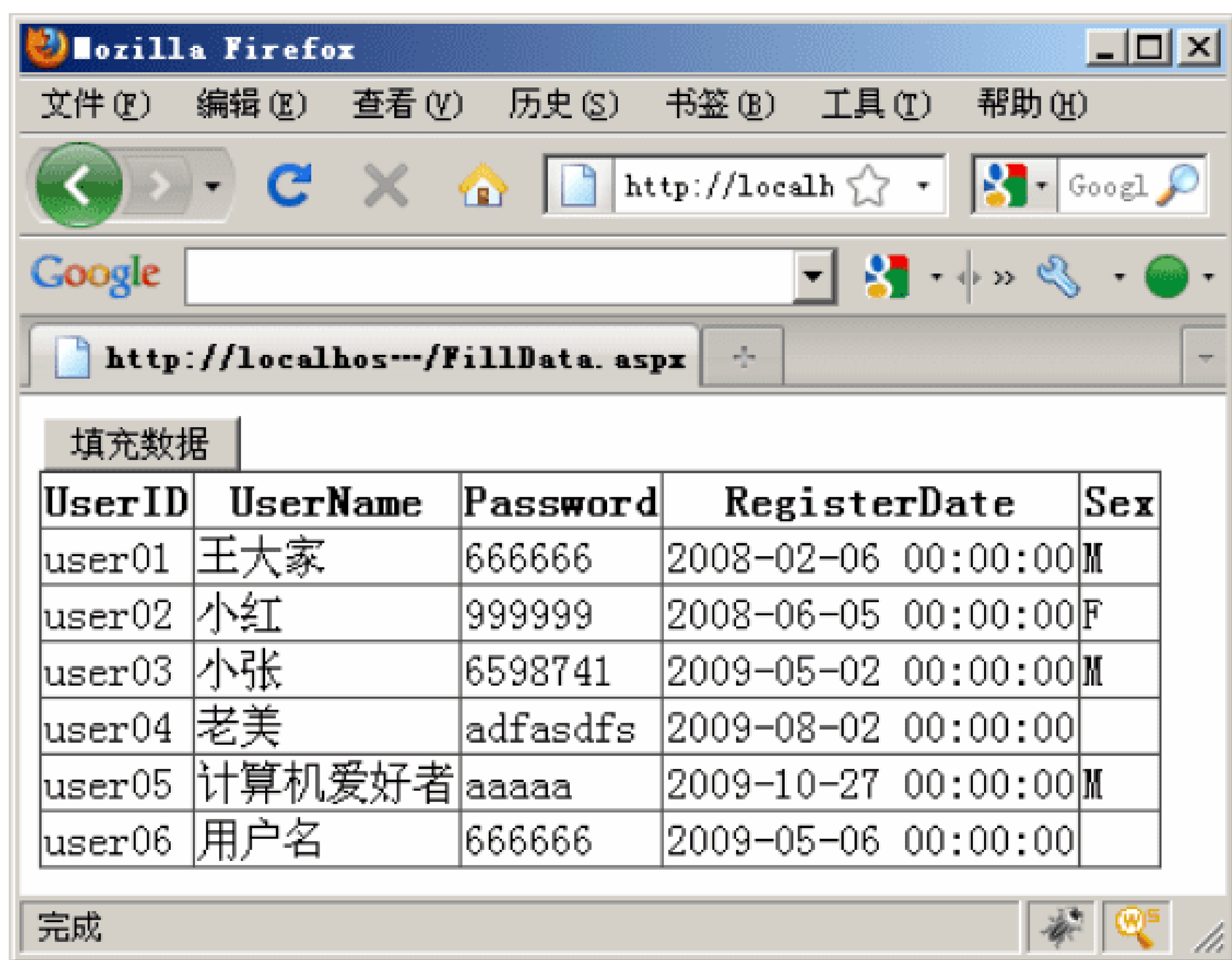
（2）在项目中添加一个页面 FillData.aspx，在页面上放置一个 Button 和一个 Table 以查询和显示数据。页面代码如下：

```
<form id="form1" runat="server">
<div>
   <asp:Button ID="Button1" runat="server" Text="填充数据" onclick=
   "Button1_Click" />
   <asp:Table ID="Table1" runat="server" BorderStyle="Solid" BorderWidth=
   "1px"
      GridLines="Both">
   </asp:Table>
</div>
</form>
```

（3）为“填充数据”按钮的 Click 事件编写如下代码。

```
protected void Button1_Click(object sender, EventArgs e)
{
   string s = System.Configuration.ConfigurationManager.ConnectionStrings
["database"].ConnectionString;
      SqlConnection connection = new SqlConnection(s);
      string sql = "select * from loginuser";
      SqlDataAdapter adapter = new SqlDataAdapter(sql, s);
                                                  //创建 DataAdapter
      DataSet ds = new DataSet();                 //创建一个 DataSet 对象
      adapter.Fill(ds);                           //填充 DataSet
      DataTable table = ds.Tables[0];
      TableHeaderRow header = new TableHeaderRow();
      //输出表头（列名）
      foreach (DataColumn column in table.Columns)
      {
         TableHeaderCell cell = new TableHeaderCell();
         cell.Text = column.ColumnName;
         header.Cells.Add(cell);
      }
      Table1.Rows.Add(header);
      //逐行输出数据
      foreach (DataRow row in table.Rows)
      {
         TableRow r = new TableRow();
         //逐列输出数据
         for (int i = 0; i < table.Columns.Count; i++)
         {
            TableCell cell = new TableCell();
            cell.Text = Convert.ToString(row[i]);
            r.Cells.Add(cell);
         }
         Table1.Rows.Add(r);
      }
      table.Dispose();
      ds.Dispose();
      adapter.Dispose();
   }
}
```

（4）运行 FillData.aspx 页面，运行界面如图 2.15 所示。


填充数据

UserID	UserName	Password	RegisterDate	Sex
user01	王大家	666666	2008-02-06 00:00:00	M
user02	小红	999999	2008-06-05 00:00:00	F
user03	小张	6598741	2009-05-02 00:00:00	M
user04	老美	adfasdfs	2009-08-02 00:00:00	
user05	计算机爱好者	aaaaa	2009-10-27 00:00:00	M
user06	用户名	666666	2009-05-06 00:00:00	

图 2.15　DataAdapter 填充数据

2.5.4　批量更新数据

DataAdapter 与数据库之间的连接是双向的，既可以从数据库读取数据，也可以向数据库更新数据。SqlDataAdaper.Update 方法的功能就是把 DataSet 中发生的变化写回到数据库中。当 SqlDataAdaper.Update 方法执行时，会检查数据集中每一行的状态（DataRowState），根据行的状态是 Added、Deleted 还是 Modified，分别调用 SqlDataAdaper 的 InsertCommand()、DeleteCommand()和 UpdateCommand()方法，实现数据的更新。如果某行的状态为 Unchanged，则不会把此行更新到数据库。

SqlDataAdapter.Update 有多个重载版本，入口参数以某种方式指定了用于更新的数据行集合，返回值是成功更新的行数。SqlDataAdapter.Update 常用有以下版本。

```
//根据 DataRow 数组中每行的状态，调用相应的命令更新数据库
public int Update(DataRow[] rows);
//更新 DataSet 中所有发生变化的行
public override int Update(DataSet dataset);
//更新 DataTable 中所有发生变化的行
public int Update(DataTable table);
//更新 DataSet 中具有指定名称的 DataTable 中发生变化的行
public int Update(DataSet dataset, string tableName);
```

SqlDataAdapter.Update()方法通过检查 DataRow 的状态判断需要对数据库执行什么操作，而不是用 DataRow 中的数据更新数据库中相应的记录。举例来说，假设数据从数据库填充到 DataSet 以后，未发生任何变化，如果人为将某一 DataRow 的状态设置为 Added，那么调用 SqlDataAdapter.Update()方法时，就会尝试把这一行插入到数据库中，虽然这一行数据并不是新增的，数据库中已经存在完全相同的记录。

在对数据库进行更新时，SqlDataAdapter 类的 UpdateCommand、InsertCommand 和 DeleteCommand 通常需要使用 DataRow 中的数据作为参数。

以插入数据为例，SqlDataAdapter 类的 InsertCommand 要把 DataRow 中各列值插入到数据库表中，所以 InsertCommand 必须带参数，具有类似以下形式。

```
insert into TableName (Column1,Column2,…) values (@value1,@value2,…);
```

其中@value1、@value2 参数的值应该来自要插入到数据库中的 DataRow 的相应列。如果不使用 SqlDataAdapter，那么应该用类似如下的代码实现插入过程。

```
//这段代码演示如何把 DataTable 的数据插入到数据库中
DataTable table;                                          //要更新的数据表
//此处省略设置 table 及其数据的代码
//insert 为要插入时执行的 SqlCommand
SqlCommand insert = new SqlCommand(
   "insert into TableName(column1,column2) values (@par1,@par2)",
   connection);
//insert 语句需要两个参数
SqlParameter par1 = new SqlParameter();
par1.ParameterName = "@par1";
insert.Parameters.Add(par1);
SqlParameter par2 = new SqlParameter();
par2.ParameterName = "@par2";
insert.Parameters.Add(par2);
//针对 DataTable 中的 DataRow 循环
foreach (DataRow row in table.Rows)
{
   //如果行状态的 Added，则设置相应的参数，执行插入命令
   if (row.RowState == DataRowState.Added)
   {
      par1.Value = row["column1"];
      par2.Value = row["column2"];
      insert.ExecuteNonQuery();
   }
}
```

SqlDataAdapter 的 Update()方法提供了批量更新数据的功能，可以用更简洁的方法实现与上述代码相同的功能。SqlParameter 有一种构造函数，可以指定参数所对应的列，从而可以在 SqlDataAdapter 的 Update()方法中自动用相应列的值作为参数的值。SqlParameter 类的可以指定参数来源的构造函数如下：

```
public SqlParameter(string paramName, SqlDbType dbType, int size, string
sourceColumn);
```

上述构造函数中，paramName 为参数名称、dbType 为参数类型、size 为参数大小、sourceColumn 为参数来源列的名称。

SqlCommandCollection 类也提供了类似的方法，如下代码所示。

```
public SqlParameter Add(string paramName, SqlDbType dbType, int size, string
sourceColumn);
```

通过使用这种自动从列中取值的参数，结合 SqlDataAdapter 类的 Update()方法，就可以用简洁的代码实现数据的批量更新。

通过为 SqlDataAdapter 设置四个命令 SelectCommand、InsertCommand、DeleteCommand 和 UpdateCommand 可以实现更新数据库，但这样做比较繁琐。当表中的列比较多时，工作量就更大。.NET Framework 中提供了一个 SqlCommandBuilder 类，可以在一定程度上减轻更新数据库时的工作量。

SqlCommandBuilder 类可以为 SqlDataAdapter 自动生成单表的更新命令。所谓单表，就是在查询和修改中只涉及一个数据库表。如果 SqlDataAdapter 的查询是多表连接查询，则 SqlCommandBuilder 不能生成任何更新命令。另外，SqlCommandBuilder 还要求在查询

中必须返回主键列或者唯一列，因为 SqlCommandBuilder 将基于这些列生成更新命令。

如果为 SqlDataAdapter 指定了 SelectCommand，并且把一个 SqlCommandBuilder 对象与 SqlDataAdapter 相关联，那么 SqlCommandBuilder 将自动生成 SqlDataAdapter 的未定义的更新命令。通过给 SqlCommandBuilder 的构造函数中传递一个 SqlDataAdapter 参数，或者设置 SqlCommandBuilder 的 DataAdapter 属性，都可以将 SqlCommandBuilder 与 SqlDataAdapter 关联起来，从而可以为 SqlDataAdapter 自动生成命令。

【例 2-7】 批量更新数据。

本例演示通过 DataAdapter 和 DataTable 向数据库批量更新数据。

（1）打开例 2-6 所创建的项目。

（2）在项目中添加一个新页面 BatchUpdate.aspx。

（3）在页面上添加一个 Table 和一个 Button。页面代码如下：

```
<form id="form1" runat="server">
<div>
<asp:Button runat="server" ID="update" Text="保存修改" onclick="update_
Click" /> <br />
<asp:Table runat="server" ID="grid" ></asp:Table>
</div>
</form>
```

（4）在页面后台的 C#代码中，添加一个页面级别的变量，存储检索到的数据。

```
private DataTable data = null;                              //查询出的数据
```

（5）在页面的 Init 事件中加载数据。代码如下：

```
protected override void OnInit(EventArgs e)
{
    base.OnInit(e);
    data = loadData();
}
private DataTable loadData()
{
   //从配置文件中获取连接字符串
    string s =
    System.Configuration.ConfigurationManager.ConnectionStrings
     ["database"].ConnectionString;
    SqlConnection connection = new SqlConnection(s);        //创建连接
    string sql = "select top 5 UserID, UserName from loginuser";
    SqlDataAdapter adapter = new SqlDataAdapter(sql, s);
                                                        //创建 DataAdapter
    DataTable table=new DataTable();                    //创建一个 DataTable
    adapter.FillSchema(table,SchemaType.Source);        //填充表架构
    adapter.Fill(table);
    adapter.Fill(table);                                //填充 DataTable
    adapter.Dispose();
    return table;
}
```

（6）在页面的 Load 事件中显示数据。代码如下：

```
protected void Page Load(object sender, EventArgs e)
 {
     showData(data);
 }
```

```
//显示 DataTable 中的数据
 private void showData(DataTable table)
 {
     int num = table.Columns.Count;                        //得到总列数
     //输出表头
     TableHeaderRow header = new TableHeaderRow();
     for (int i = 0; i < num; i++)
     {
         //在 HTML table 中循环输出各个字段名称
         TableHeaderCell cell = new TableHeaderCell();
         cell.Text = table.Columns[i].ColumnName;
         header.Cells.Add(cell);
     }
     grid.Rows.Add(header);
     //输出表中数据
     foreach (DataRow row in table.Rows)
     {
         //针对数据表中每一行数据在 HTML 表格中创建一行
         //根据数据表中每一个字段在 HTML 表格中设置一个单元格的内容
         TableRow r = new TableRow();
         string id = Convert.ToString(row[0]);             //得到用户 ID
         string name = Convert.ToString(row[1]);           //得到用户名
         TableCell cell = new TableCell();
         cell.Text = id;
         r.Cells.Add(cell);
         cell = new TableCell();
         TextBox t = new TextBox();
         t.ID = "textbox" + id;
         t.Text = name;
         t.TextChanged += new EventHandler(t TextChanged);
         cell.Controls.Add(t);
         r.Cells.Add(cell);
         grid.Rows.Add(r);
     }
 }
```

在上述代码中，用一个动态创建的 TextBox 显示用户名称。TextBox 中的文字被修改后，会触发 TextChanged 事件，在此事件中修改 DataTable 中的数据，以便存储到数据库。TextBox 控件的 TextChanged 事件处理程序如下：

```
void t_TextChanged(object sender, EventArgs e)
{
    TextBox t = sender as TextBox;
    TableRow r = t.Parent.Parent as TableRow;   //得到文本框所在行
    string id=r.Cells[0].Text;                  //得到文本框所对应的用户 ID
    //找到 DataTable 中对应此 ID 的 DataRow
    DataRow row = data.Rows.Find(id);
    if (row == null) return;
    row[1] = t.Text;                            //修改 DataRow 中的用户名
}
```

（7）在“保存修改”按钮的 Click 事件中，编写代码保存。代码如下。

```
protected void update Click(object sender, EventArgs e)
{
     //从配置文件中读取连接字符串
    string s =
    System.Configuration.ConfigurationManager.ConnectionStrings
```

```
    ["database"].ConnectionString;
    SqlConnection connection = new SqlConnection(s);        //创建连接
    string sql = "select top 5 UserID, UserName from loginuser";
                                                            //编写 SQL 语句
    SqlDataAdapter adapter = new SqlDataAdapter(sql, s);    //创建数据适配器
     //创建 CommandBuilder 对象以自动生成数据更新命令
    SqlCommandBuilder builder = new SqlCommandBuilder(adapter);
    adapter.Update(data);                                   //更新数据
    builder.Dispose();
    adapter.Dispose();
}
```

（8）运行 BatchUpdate.aspx 页面，修改多个用户名并保存，数据能够保存到数据库中。运行界面如图 2.16 所示。



图 2.16　批量更新数据

2.6　存储过程

各种大型关系数据库都支持存储过程。存储过程是一组 SQL 语句的集合，相当于 C# 语言中的方法。存储过程在效率、安全性、可复用方面比普通的 SQL 语句有优势，因此在实际项目中也有广泛应用。

2.6.1　调用存储过程

从 ADO.NET 的角度来看，存储过程也是一种数据库命令（DbCommand），只是这个 DbCommand 的 CommandType 属性为 StoredProcedure 而非默认的 Text。在 ADO.NET 中调用存储过程与执行普通的 SQL 命令方法类似。

【例 2-8】 调用存储过程修改数据。

本例将创建一个存储过程以添加一个用户，并从 ADO.NET 中调用此存储过程。

（1）创建一个 ASP.NET 应用程序，在项目 App_Data 文件夹中添加一个数据库，在数

据库中添加一个表 LoginUser，表结构如下。

UserID：登录用户 ID，nvarchar(10)类型，不可为空，主键。

UserName：用户名称，nvarchar(20)类型，不可为空。

Password：登录密码，nvarchar(20)类型，不可为空。

RegisterDate：用户注册日期，datetime 类型，可空。

Sex：性别，nchar(1)类型，可空，M 为男，F 为女。

（2）在数据库中创建一个存储过程 AddUser，这个存储过程接受用户信息作为参数，把用户信息添加到数据库中。存储过程代码如下：

```
CREATE PROCEDURE dbo.AddUser
    /*以下 5 个变量为存储过程参数*/
    @id nvarchar(10),
    @name nvarchar(20),
    @pass nvarchar(20),
    @sex nchar(1)
AS
    insert into LoginUser (UserId, UserName, Password, RegisterDate, Sex)
    values (@id, @name, @pass, getdate(), @sex)
    RETURN
```

（3）在项目中添加一个页面，在页面上放置相应控件以输入用户信息，页面代码如下：

```
<form id="form1" runat="server">
<div>
   用户登录 ID：<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
   <br />
   用户名称：<asp:TextBox ID="TextBox2" runat="server" ></asp:TextBox><br />
   登录密码：
   <asp:TextBox ID="TextBox3" runat="server" TextMode = "Password" ></asp:
   TextBox><br />
   性别：
   <asp:RadioButton ID="male" runat="server" GroupName="sex" Text="男"
   Checked="true" />
   <asp:RadioButton ID="female"  runat="server" GroupName="sex" Text=
   "女" /><br />
   <asp:Button ID="Button1" runat="server" Text="添加用户" onclick=
   "Button1_Click" /><br />
</div>
</form>
```

（4）在“添加用户”按钮的 Click 事件中编写代码。调用存储过程 AddUser 把用户数据添加到数据库，代码如下：

```
protected void Button1_Click(object sender, EventArgs e)
{
   //从配置文件读取连接字符串
   string s = ConfigurationManager.ConnectionStrings["database"].
   ConnectionString;
   SqlConnection connection = new SqlConnection(s);   //创建连接
   SqlCommand command = new SqlCommand("AddUser",connection);
   command.CommandType=CommandType.StoredProcedure;   //设置命令为存储过程
   //为存储过程添加 4 个参数
   SqlParameter p1 = new SqlParameter("@id",TextBox1.Text);
   command.Parameters.Add(p1);
   SqlParameter p2 = new SqlParameter("@name", TextBox2.Text);
```

```
    command.Parameters.Add(p2);
    SqlParameter p3 = new SqlParameter("@pass", TextBox3.Text);
    command.Parameters.Add(p3);
    SqlParameter p4=new SqlParameter("@sex",male.Checked?"M":"F");
    command.Parameters.Add(p4);
    string message="";
    try
    {
        connection.Open();                          //打开连接
        int n = command.ExecuteNonQuery();          //执行命令
        if (n == 1)
            message = "注册成功。用户信息已经保存在数据库。";
        else
            message = "数据未能保存。";
    }
    catch (Exception ex)
    {
        message = "保存用户信息过程中出错。"
            + ex.Message.Replace("\r", "\\r ").Replace("\n","\\n");
    }
    finally
    {
        command.Dispose();
        connection.Close();
    }
    Page.ClientScript.RegisterStartupScript(this.GetType(),
        "register", "<script>alert(\""+message+"\");</script>");
}
```

【例 2-9】 调用存储过程读取数据。

本例将调用存储过程，根据用户输入的关键字对用户名进行搜索，并把搜索结果显示在页面上。

（1）打开例 2-8 所创建的项目。

（2）在数据库中添加一个新的存储过程 GetUserByName，这个存储过程接受一个查找关键字，在数据库中查找用户名包含此关键字的所有用户。存储过程代码如下：

```
CREATE PROCEDURE dbo.GetUserByName
    @name nvarchar(20)
AS
    select * from LoginUser where UserName like '%'+@name+'%'
    RETURN
```

（3）在项目中添加一个新页面，在新页面上放置相应的一个 TextBox 控件以接受用户输入，放置一个服务器端 Table 以显示查询结果。页面代码如下。

```
<form id="form1" runat="server">
<div>
 输入用户名以进行搜索：<asp:TextBox runat="server" ID="userName" />
  <asp:Button ID="Button1" runat="server" Text="查找用户" onclick=
  "Button1 Click" /><br />
   查找结果如下<br />
   <asp:Table ID="result" runat="server" BorderStyle="Solid" BorderWidth=
   "1px"  GridLines="Both"/>
</div>
</form>
```

（4）在“查找用户”按钮的 Click 事件中调用存储过程执行查询，并将结果显示在页

面上的 Table 中。代码如下：

```
protected void Button1_Click(object sender, EventArgs e)
{
    //从配置文件读取连接字符串
    string s = ConfigurationManager.ConnectionStrings["database"].
    ConnectionString;
    //创建并使用数据库连接对象
    using (SqlConnection connection = new SqlConnection(s))
    {
         //创建命令对象，设置为存储过程，并添加参数
        SqlCommand command = new SqlCommand("GetUserByName", connection);
        command.CommandType = CommandType.StoredProcedure;
        SqlParameter p = new SqlParameter("@name", userName.Text);
        command.Parameters.Add(p);
        connection.Open();
        //执行命令，读取数据
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
               //将读取到的数据循环添加到 HTML 表格中
                TableRow row = new TableRow();
                result.Rows.Add(row);
                for (int i = 0; i < 3; i++)
                {
                    TableCell cell = new TableCell();
                    cell.Text = Convert.ToString(reader[i]);
                    row.Cells.Add(cell);
                }
            }
        }
        command.Dispose();
        connection.Close();
    }
}
```

（5）运行此页面，运行结果如图 2.17 所示。

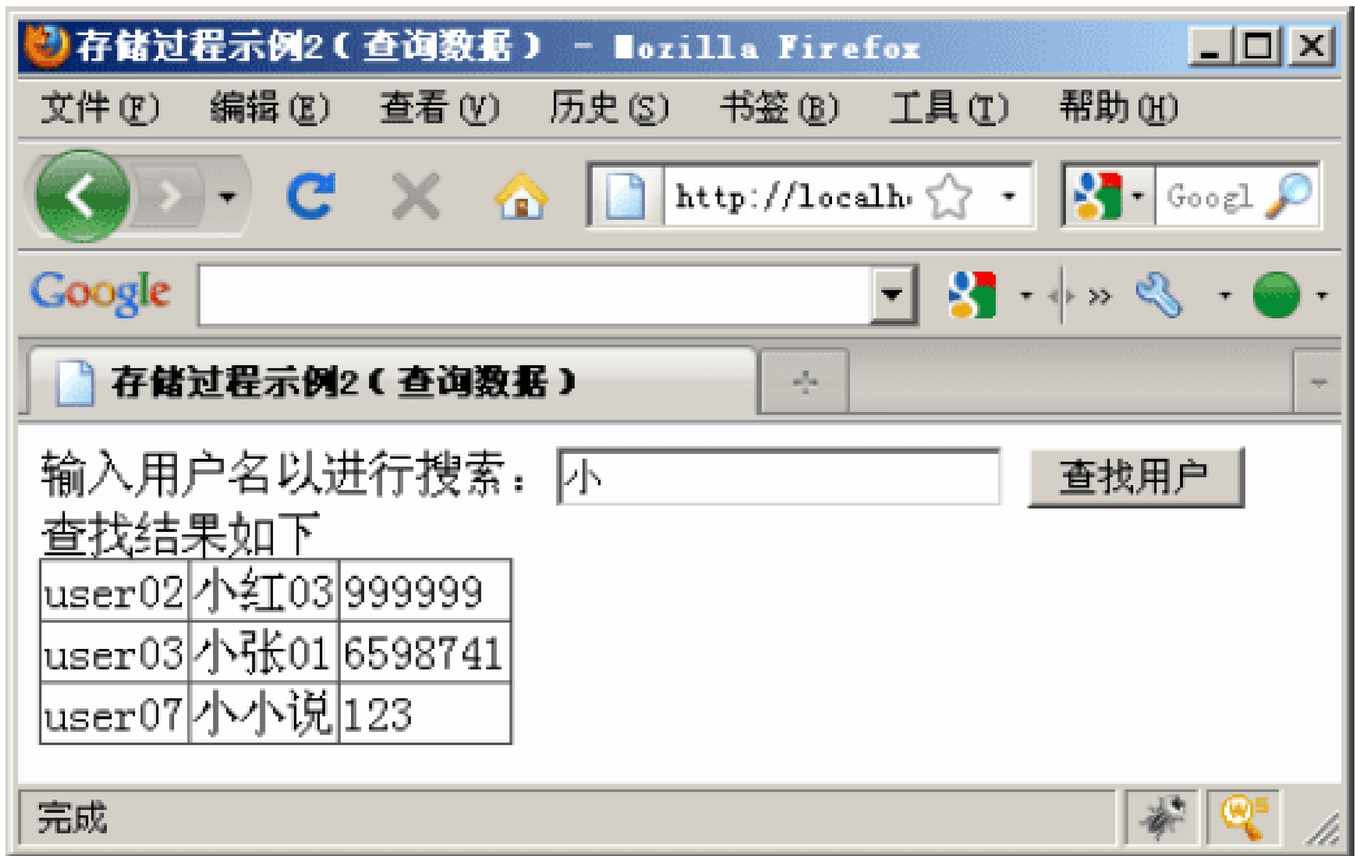


图 2.17　调用存储过程查询数据

2.6.2　输出参数

存储过程的参数默认情况下为输入参数，即参数值由存储过程调用者传递给存储过

程，如果在存储过程中对参数值进行了修改，不会影响到参数调用者。存储过程参数还有另外一种类型，称为输出参数，这种参数可以双向传值，既可以从存储过程调用者（如 ASP.NET 程序）传递给存储过程，也可以在存储过程中修改参数值，并传递回调用者。存储过程的输入输出参数的概念与 C#方法中输入输出参数概念类似。

使用存储过程的输出参数需要注意两点：一是在定义存储过程时，必须在参数后面添加 output 关键字，以说明这是一个输出参数；二是在 C#中调用带输出参数的存储过程时，必须将相应的 SqlParameter 参数的 Direction 属性设置为 ParameterDirection.Output。

【例 2-10】 存储过程输出参数。

本例将查询所有注册用户总数、男用户数和女用户数，利用存储过程输出参数来实现。

（1）打开例 2-8 所创建的项目。

（2）在数据库中添加一个新的存储过程 GetUserCount，代码如下：

```
CREATE PROCEDURE dbo.GetUserCount
    @total int output,
    @male int output,
    @female int output
AS
    select @total=count(*) from LoginUser ;
    select @male=count(*) from LoginUser where Sex='M';
    select @female=count(*) from LoginUser where Sex='F';
    RETURN
```

（3）在项目中添加一个新页面 OutputParameter.aspx，在页面上放置 1 个 Button 以调用存储过程，添加 3 个 Label 以显示查询结果。页面代码如下：

```
<form id="form1" runat="server">
<div>
<asp:Button ID="Button1" runat="server" Text="统计用户数" onclick=
"Button1_Click" /><br />
用户总数为<asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>,
其中男<asp:Label ID="Label2" runat="server" Text="Label"></asp:Label>,
女<asp:Label ID="Label3" runat="server" Text="Label"></asp:Label>, <br />
其余用户注册时未说明性别。
</div>
</form>
```

（4）在“统计用户数”按钮的 Click 事件中，编写代码以执行存储过程并显示结果，代码如下：

```
protected void Button1_Click(object sender, EventArgs e)
{
    string s = ConfigurationManager.ConnectionStrings["database"]
    .ConnectionString;
    using (SqlConnection connection = new SqlConnection(s))
    {
        //创建一个命令对象，设置命令类型为存储过程
        SqlCommand command = new SqlCommand("GetUserCount", connection);
        command.CommandType = CommandType.StoredProcedure;
        //创建参数，并说明参数类型
        SqlParameter p1 = new SqlParameter("@total",SqlDbType.Int);
        p1.Direction = ParameterDirection.Output;   //定义为输出参数
        command.Parameters.Add(p1);                 //把参数添加到存储过程
        //以同样的方式为存储过程添加其他输出参数
```

```
            SqlParameter p2 = new SqlParameter("@male", SqlDbType.Int);
            p2.Direction = ParameterDirection.Output;
            command.Parameters.Add(p2);
            SqlParameter p3 = new SqlParameter("@female", SqlDbType.Int);
            p3.Direction = ParameterDirection.Output;
            command.Parameters.Add(p3);
            connection.Open();
            command.ExecuteNonQuery();                  //执行命令
            //命令执行完毕后，命令的输出参数中已经包含结果
        connection.Close();
        command.Dispose();
        //取得输出参数的值
        int n1 = (int)command.Parameters["@total"].Value;
        int n2 = (int)command.Parameters["@male"].Value;
        int n3 = (int)command.Parameters["@female"].Value;
        //把结果显示在页面控件上
        Label1.Text = n1.ToString();
        Label2.Text = n2.ToString();
        Label3.Text = n3.ToString();
    }
}
```

2.7　事　　务

事务是数据库中一个很重要的概念，是作为单个逻辑工作单元执行的一系列操作。事务具有 4 个重要属性，原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation，又称独立性）、持久性（Durability），事务的这 4 个属性也可以合并称为 ACID。

2.7.1　事务的基本概念

事务有以下 4 个基本属性。

- 原子性：事务必须是原子工作单元；对于其数据修改，或者全都执行，或者全都不执行。
- 一致性：事务在完成时，必须使所有的数据都保持一致状态。在相关数据库中，所有规则都必须应用于事务的修改，以保持所有数据的完整性。事务结束时，所有的内部数据结构都必须是正确的。
- 隔离性：由并发事务所作的修改必须与任何其他并发事务所作的修改隔离。事务识别数据时数据所处的状态，或者是另一并发事务修改之前的状态，或者是第二个事务修改之后的状态，事务不会识别中间状态的数据。
- 持久性：事务完成之后，对于系统的影响是永久性的。该修改即使出现系统故障也将一直保持。

为了理解事务的概念，一个常用的例子就是银行转账。在银行转账过程中，从原账户扣除金额，向目标账户添加金额，这两个数据库操作的总和构成一个完整的逻辑过程，不可拆分。这个过程被称为一个事务，具有 ACID 特性。

SQL Server 中主要有 3 处事务：显式事务、自动提交事务和隐式事务，其中显式事务

最常使用，而且其操作比较典型，本节将以显式事务为例介绍事务的使用。

事务具有开始和结束。在 SQL Server 中，以显式事务为例，事务从 Begin Transaction 语句作为开始。事务的结束又可分为两种情况，分别称为提交（Commit）和回滚（Rollback）。提交事务是指让事务中所有操作生效，而回滚事务是指取消事务中所有操作，使数据库恢复到事务开始前的状态。

事务可以嵌套，即在一个事务中还可以再开始一个事务。

提示： 事务的提交与事务嵌套的层次是相对应的，即 Commit 语句只提交嵌套事务中该层次的事务，而 Rollback 语句会回滚所有事务，不论 Rollback 语句处于嵌套事务哪个层次。

【例 2-11】 在 SQL Server 中使用事务。

本例演示在 SQL Server 中开始事务、提交事务、回滚事务，以及嵌套事务。

（1）打开 SQL Server 2005 Management Studio，新建一个数据库 Test。

（2）在数据库中新建一个商品表 Products，创建表的 SQL 代码如下。

```
CREATE TABLE [dbo].[Products](
    [ProductID] [nchar](10) NOT NULL,          /*商品编号*/
    [ProductName] [nvarchar](20) NOT NULL,     /*商品名称*/
    [Price] [float] NOT NULL,                  /*商品价格*/
    [Stock] [float] NOT NULL,                  /*商品库存*/
 CONSTRAINT [PK_Products] PRIMARY KEY CLUSTERED
([ProductID] ASC) )
```

（3）向 Products 表中添加几行数据，SQL 代码如下：

```
insert into Products (productid, productname, price, stock)
values ('p001','CPU',425,6);
insert into Products (productid, productname, price, stock)
values ('p002','内存',180,20);
insert into Products (productid, productname, price, stock)
values ('p003','显示器',1100,5);
```

（4）下面演示事务和回滚，在 SQL Server 2005 中编写以下 SQL 语句。

```
set nocount on
begin  transaction  /*开始事务*/
update Products set price=9999 where ProductID='p001'
delete Products where ProductID='p002'
print 'Products 数据为（在事务中）'
select * from Products
rollback transaction  /*回滚事务*/
print 'Products 数据为（事务回滚后）'
select * from Products
```

运行以下语句，得到结果如下：

```
Products 数据为（在事务中）
ProductID  ProductName         Price                Stock
---------- ------------------- -------------------- ----------------------
p001       CPU                 9999                 6
p003       显示器              1100                 5
Products 数据为（事务回滚后）
```

```
ProductID  ProductName         Price                Stock
---------- ------------------- -------------------- --------------------
p001       CPU                 425                  6
p002       内存                180                  20
p003       显示器              1100                 5
```

从以上运行结果可以看出，在事务中，把 CPU 的价格进行了调整，而且删除了 P002 产品。后来由于事务回滚，所做修改全部撤销，数据又恢复到事务开始前的状态。

（5）下面演示事务和提交。在 SQL Server 2005 中编写以下 SQL 语句。

```
set nocount on
begin  transaction  /*开始事务*/
update Products set price=9999 where ProductID='p001'
print 'Products 数据为（在事务中）'
select * from Products where ProductID='p001'
commit transaction  /*提交事务*/
print 'Products 数据为（事务提交后）'
select * from Products where ProductID='p001'
```

运行以上语句，执行结果如下：

```
Products 数据为（在事务中）
ProductID  ProductName         Price                Stock
---------- ------------------- -------------------- --------------------
p001       CPU                 9999                 6
Products 数据为（事务提交后）
ProductID  ProductName         Price                Stock
---------- ------------------- -------------------- --------------------
p001        CPU                 9999                6
```

从以上运行结果看，由于事务进行了提交，所以事务中所做的修改（CPU 价格改为 9999）得以保存。

（6）下面演示嵌套事务。在 SQL Server 2005 中编写以下代码：

```
set nocount on
print '未修改以前的原始数据'
select * from Products where ProductID='p001'
begin  transaction   /*开始事务*/
update Products set price=666 where ProductID='p001'
print 'Products 数据为（在顶层事务中）'
select * from Products where ProductID='p001'
begin transaction    /*开始嵌套事务*/
update Products set Stock=100 where ProductID='p001'
print 'Products 数据为（在嵌套事务中）'
select * from Products where ProductID='p001'
commit transaction  /*提示嵌套事务*/
print 'Products 数据为（嵌套事务提交后）'
select * from Products where ProductID='p001'
rollback transaction  /*回滚顶层事务*/
print 'Products 数据为（顶层事务回滚后）'
select * from Products where ProductID='p001'
```

以上代码中有两个嵌套事务，在顶层事务中把商品 P001 价格修改为 666，在嵌套事务中把商品 P001 的数量修改为 100，然后提交嵌套事务，回滚顶层事务，代码运行结果如下：

```
未修改以前的原始数据
ProductID  ProductName        Price                Stock
---------- ------------------ -------------------- --------------------
p001       CPU                888                  6
Products 数据为（在顶层事务中）
ProductID  ProductName        Price                Stock
---------- ------------------ -------------------- --------------------
p001       CPU                666                  6
Products 数据为（在嵌套事务中）
ProductID  ProductName        Price                Stock
---------- ------------------ -------------------- --------------------
p001       CPU                666                  100
Products 数据为（嵌套事务提交后）
ProductID  ProductName        Price                Stock
---------- ------------------ -------------------- --------------------
p001       CPU                666                  100
Products 数据为（顶层事务回滚后）
ProductID  ProductName        Price                Stock
---------- ------------------ -------------------- --------------------
p001       CPU                888                  6
```

从上述运行结果可以看出，虽然嵌套事务被提交，但是由于顶层事务被回滚，所以嵌套事务中所做的修改也没有保存。

2.7.2　ADO.NET 中的事务

在 ADO.NET 中，用 System.Data.Common.DbTransaction 类表示数据库的事务。DbTransaction 类是一个抽象类，不能被实例化。DbTransaction 类作为其他具体数据库事务类的基类，定义了公共接口。对于 SQL Server 数据库来说，SqlTransaction 表示数据库的事务。

DbConnection 常用有以下两个方法。

- Commit()：提交数据库事务。此方法无参数和返回值。
- Rollback()：回滚数据库事务。此方法无参数和返回值。

DbConnection 及其派生类没有公共构造函数，所以不能用 new 创建一个 DbTransaction 或其派生类的实例。要想得到 DbTransaction 的一个实例，应该调用 DbConnection 类的 BeginTransaction()方法。DbConnection 类的 BeginTransaction()方法签名如下。

```
public DbTransaction BeginTransaction();
```

一个数据库的事务中通常包含多个操作。对于 ADO.NET 来说，一个数据库事务通常包含多个数据库命令 DbCommand 对象。为了将 DbCommand 与事务 DbTransaction 关联起来，需要设置 DbCommand 类的 Transaction 属性。DbCommand 类的 Transaction 属性签名如下：

```
public DbTransaction Transaction { get; set; }
```

【例 2-12】　ADO.NET 事务。

本例演示如何在 ADO.NET 中使用数据库事务，包含开始事务、提交事务、回滚事务。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加一个 LoginUser 表，表结构同上一节的例子。

（3）在项目中添加一个页面 AdoNetTran.aspx，页面上放置一个 Button，在 Button 的 Click 事件中，添加如下代码：

```
protected void Button1 Click(object sender, EventArgs e)
{
     //从配置文件读取连接字符串
    string s = ConfigurationManager.ConnectionStrings["database"].
    ConnectionString;
    SqlConnection connection = new SqlConnection(s);        //创建连接
    //创建一个命令对象
    SqlCommand command1 =
       new SqlCommand("insert into LoginUser (UserID, UserName, Password,
       Sex) "+" values ('user99','新用户','123','M') ", connection);
    connection.Open();
   //创建另外一个命令对象
    SqlCommand command2=
       new SqlCommand("insert into LoginUser (UserID,UserName,Password,
       Sex) "+" values ('user100','新用户','123','男生') ", connection);
     //上面这条语句会出错，因为性别"男生"为两个字符，超出字段宽度
    SqlTransaction tran=connection.BeginTransaction(); //开始一个事务
    //将以上两个命令添加到同一个事务中
    command1.Transaction=tran;
    command2.Transaction=tran;
    string message="";
    try
    {
        command1.ExecuteNonQuery();                          //执行命令 1
        command2.ExecuteNonQuery();                          //执行命令 2
        tran.Commit();  //提交事务
        message="保存成功！";
    }
    catch (Exception ex)
    {
        message = "保存用户信息过程中出错。事务回滚。" ;
        tran.Rollback();                                     //发生异常时事务回滚
    }
    finally
    {
        command1.Dispose();
        command2.Dispose();
        tran.Dispose();
        connection.Close();
    }
    Page.ClientScript.RegisterStartupScript(this.GetType(),
        "register", "<script>alert(\""+message+"\");</script>");
}
```

在上面的代码中，一共执行了两条插入命令，向 LoginUser 表中插入两个新用户，这两个命令使用同一个事务。第一条命令是正确的，而第二条命令所插入的数据不符合数据库定义（性别字段太长超出字段宽度）在执行时会发生异常。程序在运行时会从第二条命令执行的语句跳转到 catch 块中，回滚事务。由于两个命令使用同一个事务，事务一旦回滚，两个命令所做的改动都被撤销，数据库恢复成命令执行前的状态。

2.7.3　TransactionScope 类的使用

.NET 中还有另外一种处理事务的方式：使用 System.Transactions.TransactionScope 类。TransactionScope 类在 System.Transactions 程序集中，默认情况下，ASP.NET 应用程序并不包含对这个程序集的引用，所以如果使用 TransactionScope 类，就需要手工添加对 System.Transactions 程序集的引用。

2.7.2 节所讲的 DbTransaction 类只能用于一个数据库事务处理，而 TransactionScope 类可用于分布式多个数据库事务。利用 TransactionScope 类能够创建可提升事务。可提升事务是可以根据需要自动提升为完全分布式事务的轻型（本地）事务。

TransactionScope 类只有一个常用方法 Complete()方法，指示事务正常结束。如果不调用 Complete()方法，那么当 TransactionScope 对象释放时，将会回滚事务中的所有操作。

TransactionScope 类表示的事务范围为从创建 TransactionScope 类的实例开始，到调用 Complete()方法或者这个 TransactionScope 类的实例被释放。

通常用一条 using 语句来使用 TransactionScope 类时，把事务中包含的语句放到 using 的大括号中，在大括号内最后一条语句是 TransactionScope.Complete()方法。代码如下：

```
using (TransactionScope scope = new TransactionScope())
{
    //这里写执行事务中的操作
    scope.Complete();  //结束事务
}
```

在上述代码中，如果大括号中的语句都执行成功（未发生异常），那么 TransactionScope.Complete()方法就会被调用，事务提交，否则，如果大括号中任意一条语句发生异常，那么程序就会跳到 catch 中执行（有 catch 语句时）或者中断（没有 catch 语句时），TransactionScope.Complete()方法不会被调用，事务回滚。

【例 2-13】 TransactionScope 示例。

本例将向两个数据库中添加数据，并且通过 TransactionScope 类把这两个不同数据库的操作合并为一个事务来处理，统一提交或者回滚。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加对 System.Transactions 程序集的引用。

（3）在项目中添加两个数据库，名称分别为 database1.mdf 和 database2.mdf。在两个数据库中分别创建一个 Products 表，建表 SQL 语句如下：

```
CREATE TABLE [Products](
    [ProductID] [nchar](10) NOT NULL,      /*商品编号*/
    [ProductName] [nvarchar](20) NOT NULL, /*商品名称*/
    [Price] [float] NOT NULL,              /*商品价格*/
    [Stock] [float] NOT NULL,              /*商品库存*/
 CONSTRAINT [PK_Products] PRIMARY KEY CLUSTERED
([ProductID] ASC) )
```

（4）在项目中添加一个页面 TransactionScopePage.aspx，在页面上放一个 Button。在 Button 的 Click 事件中编写代码，向两个数据库的 Products 表添加数据，并使用 TransactionScope 类将两个不同数据库的插入命名组合为一个事务。代码如下：

```
protected void Button1 Click(object sender, EventArgs e)
{
    //两个连接字符串，连接到两个不同数据库
    string s1 = @"Data Source=.\SQLEXPRESS;AttachDbFilename=
    |DataDirectory|\Database1.mdf;Integrated Security=True;User Instance=
    True";
    string s2 = @"Data Source=.\SQLEXPRESS;AttachDbFilename=
    |DataDirectory|\Database2.mdf;Integrated Security=True;User Instance=
    True";
    //使用以上两个连接字符串创建两个连接
    SqlConnection connection1 = new SqlConnection(s1);
    SqlConnection connection2 = new SqlConnection(s2);
    //创建一个命令对象执行一条 insert 语句
    SqlCommand command = new SqlCommand( );
 command.CommandText="insert into Products (ProductID,ProductName,Price,
Stock) "+" values ('P101','新产品',100,20) ";
    using(TransactionScope tran=new TransactionScope())
    {
        string message="";
        try
        {
            //让命令使用第一个连接，更新第一个数据库
            connection1.Open();
            command.Connection = connection1;
            command.ExecuteNonQuery();                       //执行命令
            //让命令使用第二个连接，更新第二个数据库
            connection2.Open();
            command.Connection = connection2;
            command.ExecuteNonQuery();                       //执行命令
            message="保存成功! ";
            tran.Complete();                                 //提交事务
        }
        catch (Exception ex)
        {
            //如果执行到 catch 块中，说明 try 中有异常
            //而 try 的最后一句 tran.Complete 肯定没有成功执行，所以事务没有提交
            message = "保存用户信息过程中出错。事务回滚。" ;
        }
        finally
        {
            command.Dispose();
            tran.Dispose();
            //检查两个连接状态，如果打开则关闭
            if(connection1.State==ConnectionState.Open)
                connection1.Close();
            if (connection2.State == ConnectionState.Open)
                connection2.Close();
        }
        Page.ClientScript.RegisterStartupScript(this.GetType(),
            "register", "<script>alert(\""+message+"\");</script>");
    }
}
```

（5）运行页面，单击按钮，可以看到，上述代码成功将数据插入到两个数据库中。

（6）修改上述代码，将第二条命令文本改成一条错误 SQL 语句，从而导致程序运行时出现异常。关键代码如下：

```
try
{
    //让命令使用第一个连接，更新第一个数据库
    connection1.Open();
    command.Connection = connection1;
    command.ExecuteNonQuery();                          //执行命令
    //让命令使用第二个连接，更新第二个数据库
    command.CommandText = "insert into Products (price) values (3)";
    connection2.Open();
    command.Connection = connection2;
    command.ExecuteNonQuery();                          //执行命令
    message="保存成功！";
    tran.Complete();                                    //提交事务
}
```

（7）再次运行页面，单击按钮，可以看到，由于代码出错，事务回滚，两条修改数据的语句都被撤销。

2.8　通用数据访问类 SqlHelper

通过前面几节的介绍，可以看到用 ADO.NET 操作数据库存在大量的重复工作。一般来说，ADO.NET 操作数据库包含以下几个步骤：

（1）获取连接字符串，创建到数据库的连接。

（2）打开连接。

（3）创建命令对象。

（4）为命令添加参数。

（5）执行命令，获得命令结果。

（6）关闭命令。

（7）关闭连接。

（8）处理命令结果。

在上述步骤中，除了命令文本不同，执行命令的方法和返回内容不同，其他步骤都是相同的，甚至代码都一样。从软件设计的角度来说，应该设计一个通用数据类，负责大部分的重复工作，从而避免每次访问数据库都要写大量重复代码。实际上，ADO.NET 刚出现时，就有开发人员写出了这样的通用数据访问类，包括微软公司自己也推出了通用数据访问类 SqlHelper 类。微软公司的通用数据访问类经过不断完善发展，现在已经成为了企业库 EnterpriseLibrary 的一部分。

本节将介绍 SqlHelper 类的设计和实现。这个 SqlHelper 类的代码主要来源网络，笔者对其进行了适当修改。从尊重知识产权角度来说，笔者需要给出 SqlHelper 类的原始作者和链接，但是由于 SqlHelper 类在网上有很多转载和不同版本，笔者无法找到原始作者，所以这里无法注明出处。

从前面分析的 ADO.NET 操作数据库的 8 个步骤来看，主要涉及 3 种对象：数据库连接、数据库命令、命令参数。通用数据访问类的主要功能就是管理这 3 种对象。

2.8.1　管理连接

数据库连接是 ADO.NET 中最基础的对象，所有数据库操作都要基于一个数据库连接。通用数据访问类中对数据库连接的管理主要就是获取连接字符串和创建连接。相关代码：

```
public class SqlHelper
{
    //数据提供程序
    private static string dbProviderName =
        ConfigurationManager.ConnectionStrings ["database"].Provider
        Name;
    //连接字符串
    private static string dbConnectionString =
        ConfigurationManager.ConnectionStrings ["database"].Connection
        String;
    private DbConnection connection;                //连接对象
    #region 构造函数
    public SqlHelper()
    {
        this.connection = CreateConnection(SqlHelper.dbConnectionString);
    }
    public SqlHelper(string connectionString)
    {
        this.connection = CreateConnection(connectionString);
    }
    #endregion
    #region 创建连接
    /// <summary>
    /// 根据配置文件中的连接字符串创建数据库连接对象
    /// </summary>
    /// <returns>所创建的连接对象</returns>
    public static DbConnection CreateConnection()
    {
        DbProviderFactory dbfactory = DbProviderFactories. GetFactory
        ( SqlHelper.dbProviderName);
        DbConnection dbconn = dbfactory.CreateConnection();
        dbconn.ConnectionString = SqlHelper.dbConnectionString;
        return dbconn;
    }
    /// <summary>
    /// 用指定的连接字符串创建数据库连接对象
    /// </summary>
    /// <param name="connectionString">连接字符串</param>
    /// <returns>所创建的连接对象</returns>
    public static DbConnection CreateConnection(string connectionString)
    {
        DbProviderFactory dbfactory = DbProviderFactories.GetFactory
         (SqlHelper.dbProviderName);
        DbConnection dbconn = dbfactory.CreateConnection();
        dbconn.ConnectionString = connectionString;
        return dbconn;
    }
    #endregion
...
}
```

2.8.2　创建命令

创建好数据库连接以后，接下来的工作就是创建数据库命令。数据库命令主要分为两种：文本命令和存储过程。SqlHelper 类定义了两个方法，分别用于创建这两种命令。

```
#region 创建命令
/// <summary>
/// 创建存储过程命令
/// </summary>
/// <param name="storedProcedure">存储过程名称</param>
/// <returns>所创建的命令</returns>
public DbCommand GetStoredProcCommond(string storedProcedure)
{
    DbCommand dbCommand = connection.CreateCommand();
    dbCommand.CommandText = storedProcedure;
    dbCommand.CommandType = CommandType.StoredProcedure;
    return dbCommand;
}
/// <summary>
/// 创建文本命令
/// </summary>
/// <param name="sql">命令文本</param>
/// <returns>所创建的命令</returns>
public DbCommand GetSqlStringCommond(string sql)
{
    DbCommand dbCommand = connection.CreateCommand();
    dbCommand.CommandText = sql;
    dbCommand.CommandType = CommandType.Text;
    return dbCommand;
}
#endregion
```

2.8.3　添加命令参数

ADO.NET 中的命令通常都带有一个或者多个参数。在原始的 ADO.NET 代码中，创建和添加命令参数也是一项很繁琐的工作，有必要在 SqlHelper 类中添加相应的方法实现这个功能。在 ADO.NET 中，数据库命令参数分为两种：输入参数和输出参数，在编写方法时也要考虑到这两点。

```
/// <summary>
/// 为命令添加输出参数
/// </summary>
/// <param name="cmd">命令</param>
/// <param name="parameterName">参数名</param>
/// <param name="dbType">参数类型</param>
/// <param name="size">大小</param>
public void AddOutParameter(DbCommand cmd, string parameterName, DbType
dbType, int size)
{
    DbParameter dbParameter = cmd.CreateParameter();
    dbParameter.DbType = dbType;
    dbParameter.ParameterName = parameterName;
```

```
    dbParameter.Size = size;
    dbParameter.Direction = ParameterDirection.Output;
    cmd.Parameters.Add(dbParameter);
}
/// <summary>
/// 为命令添加输入参数
/// </summary>
/// <param name="cmd">命令</param>
/// <param name="parameterName">参数名</param>
/// <param name="dbType">参数类型</param>
/// <param name="value">参数值</param>
public void AddInParameter(DbCommand cmd, string parameterName, DbType
dbType, object value)
{
    DbParameter dbParameter = cmd.CreateParameter();
    dbParameter.DbType = dbType;
    dbParameter.ParameterName = parameterName;
    dbParameter.Value = value;
    dbParameter.Direction = ParameterDirection.Input;
    cmd.Parameters.Add(dbParameter);
}
/// <summary>
/// 根据参数名得到参数
/// </summary>
/// <param name="cmd">命令</param>
/// <param name="parameterName">参数名</param>
/// <returns>得到的参数</returns>
public DbParameter GetParameter(DbCommand cmd, string parameterName)
{
    return cmd.Parameters[parameterName];
}
```

2.8.4　执行命令

正如本章前面几节所介绍的，在 ADO.NET 中，数据库命令根据查询结果不同，有几种不同的执行方法，分别是没有查询结果的 ExecuteNonQuery()方法、查询单个值的 ExecuteScalar()方法和查询多行多列的 ExecuteReader()方法。通用数据访问类 SqlHelper 也是按照这种分类方式对数据库命令的执行进行了封装。主要代码如下：

```
//执行命令，得到 DataReader 对象
public DbDataReader ExecuteReader(DbCommand cmd)
{
    cmd.Connection.Open();
    DbDataReader reader = cmd.ExecuteReader(CommandBehavior.Close
    Connection);
    return reader;
}
//执行没有查询结果的命令（如 insert，delete），返回受影响的行数
public int ExecuteNonQuery(DbCommand cmd)
{
    cmd.Connection.Open();
    int ret = cmd.ExecuteNonQuery();
    cmd.Connection.Close();
    return ret;
}
```

```
//执行命令，并返回查询到的单个值
public object ExecuteScalar(DbCommand cmd)
{
    cmd.Connection.Open();
    object ret = cmd.ExecuteScalar();
    cmd.Connection.Close();
    return ret;
}
/// <summary>
/// 执行命令，返回一个 DataTable
/// </summary>
/// <param name="cmd">要执行的命令</param>
/// <returns>得到的 DataTable</returns>
public DataTable ExecuteDataTable(DbCommand cmd)
{
    DbDataReader reader = ExecuteReader(cmd);
    DataTable table = new DataTable();
    table.Load(reader);
    reader.Close();
    return table;
}
```

2.8.5　释放资源

数据库连接是一种宝贵资源，使用完毕后应该及时释放。对于 SqlHelper 类来说，应该提供一个显式的释放数据库连接的接口以供用户调用，同时还应该在垃圾回收时自动释放数据库连接。可以通过 IDisposable 接口来实现上述功能。

IDisposable 接口中只包含一个方法 Dispose()，在此方法中完成对当前对象所占用资源的释放。在实现 IDisposable 接口时，要考虑到两种不同的情况，即用户代码调用 Dispose 方法释放资源和垃圾回收时释放资源。如果用户代码调用 Dispose()方法，则此时对象没有被垃圾回收，对象中的所有资源（包括托管资源和非托管资源）都是可用的，应该全部释放这些资源。而当对象被垃圾回收时，对象中的托管资源已经被释放，则只需要释放非托管资源。根据以上讨论，可以对 SqlHelper 类作以下修改。

```
public class SqlHelper:IDisposable
{
    #region 释放资源
    private bool disposed = false;                      //资源是否已经被释放
    #region IDisposable 成员
    public void Dispose()
    {
        dispose(true);
    }
    /// <summary>
    /// 释放资源
    /// </summary>
    /// <param name="disposing">是否用户代码调用</param>
    /// <remarks>
    /// 此方法将在两种情况下被调用，（1）用户代码调用 Dispose 方法()（2）垃圾回收
    /// 在第 2 种情况下，数据库连接已经被垃圾回收自动处理，不需要再释放
    /// 只有在第 1 种情况下需要在代码中显式关闭数据库连接
    /// </remarks>
```

```
    private void dispose(bool disposing)
    {
        if (disposed) return;
        disposed = true;
        if(disposing)
            if (connection.State != ConnectionState.Closed)
                connection.Close();
    }
    #endregion
    #endregion
}
```

2.8.6　SqlHelper 应用举例

前面介绍了 SqlHelper 类的设计和实现，下面通过一个实例来说明其应用。这个例子的功能与前面所讲的两个例子功能相同，但实现方法不一样。对照这两种不同的实现，更加能够体现 SqlHelper 类所带来的方便。

【例 2-14】 SqlHelper 示例。

本例演示如何使用通用数据访问类 SqlHelper。本例包括两个功能：查询用户列表和添加新用户。其中添加用户功能通过存储过程实现，而查询用户通过 SQL 文本命令实现。

（1）创建一个 ASP.NET 应用程序。

（2）在项目的 App_Data 文件夹中添加一个数据库，向数据库中添加一个 LoginUser 表，表结构同前。

（3）在数据库中添加一个存储过程 AddUser。

```
CREATE PROCEDURE dbo.AddUser
    @id nvarchar(10),  @name   nvarchar(20),@pass   nvarchar(20),@sex
    nchar(1)
AS
    insert into LoginUser (UserId, UserName, Password, RegisterDate, Sex)
    values (@id, @name, @pass, getdate(), @sex)
```

（4）在项目中添加一个页面 SqlHelperDemo.aspx。页面上部放置一个 Table，用于显示用户列表。页面下部放置几个 TextBox 和 RadioButton，用于输入用户信息以添加用户。页面代码如下：

```
<form id="form1" runat="server">
<div>
<div>
<h3>用户列表</h3>
    <asp:Table ID="result" runat="server" GridLines="Both" CellPadding=
    "2">
    </asp:Table>
</div>
<hr />
<div>
    <h3>添加新用户</h3>
    用户登录 ID：<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
    <br />
    用户名称：<asp:TextBox ID="TextBox2" runat="server" ></asp:TextBox><br />
    登录密码：
    <asp:TextBox ID="TextBox3" runat="server" TextMode="Password"></asp:
```

```
    TextBox><br />
    性别：
    <asp:RadioButton ID="male" runat="server" GroupName="sex" Text="男"
    Checked="true" />
    <asp:RadioButton ID="female"  runat="server" GroupName="sex" Text=
    "女" /><br />
    <asp:Button ID="Button1" runat="server" Text="添加用户" onclick=
    "Button1_Click" /><br />
</div>
</div>
</form>
```

（5）在 Page_Load 事件中，编写代码加载用户信息并显示。代码如下：

```
protected void Page_Load(object sender, EventArgs e)
{
    if(!IsPostBack)
        showUserList();
}
//从数据库中查询所有用户数据并显示在页面上
private void showUserList()
{
    SqlHelper db = new SqlHelper();                //创建 SqlHelper 实例
    //通过 SqlHelper 创建命令对象
    DbCommand command = db.GetSqlStringCommond("Select * from LoginUser");
    //通过 SqlHelper 执行命令并获得数据读取器
    using (DbDataReader reader = db.ExecuteReader(command))
    {
        //将读取到的数据循环显示在页面上
        while (reader.Read())
        {
            TableRow row = new TableRow();
            result.Rows.Add(row);
            for (int i = 0; i < 5; i++)
            {
                TableCell cell = new TableCell();
                cell.Text = Convert.ToString(reader[i]);
                row.Cells.Add(cell);
            }
        }
    }
    db.Dispose();
}
```

（6）在“添加用户”按钮的 Click 事件中，编写代码完成添加用户操作。代码如下：

```
protected void Button1_Click(object sender, EventArgs e)
{
    SqlHelper db = new SqlHelper();                //创建 SqlHelper 实例
    DbCommand command = db.GetStoredProcCommond("AddUser");
                                                   //得到存储过程命令
    db.AddInParameter(command, "@id", DbType.String, TextBox1.Text);
                                                   //向存储过程添加参数
    db.AddInParameter(command, "@name", DbType.String, TextBox2.Text);
    db.AddInParameter(command, "@pass", DbType.String, TextBox3.Text);
    db.AddInParameter(command, "@sex",DbType.String, male.Checked ? "M" :
    "F");
    db.ExecuteNonQuery(command);                   //执行命令
    db.Dispose();
```

```
    showUserList();                                           //刷新用户列表
}
```

（7）运行 SqlHelperDemo.aspx 页面，运行界面如图 2.18 所示。

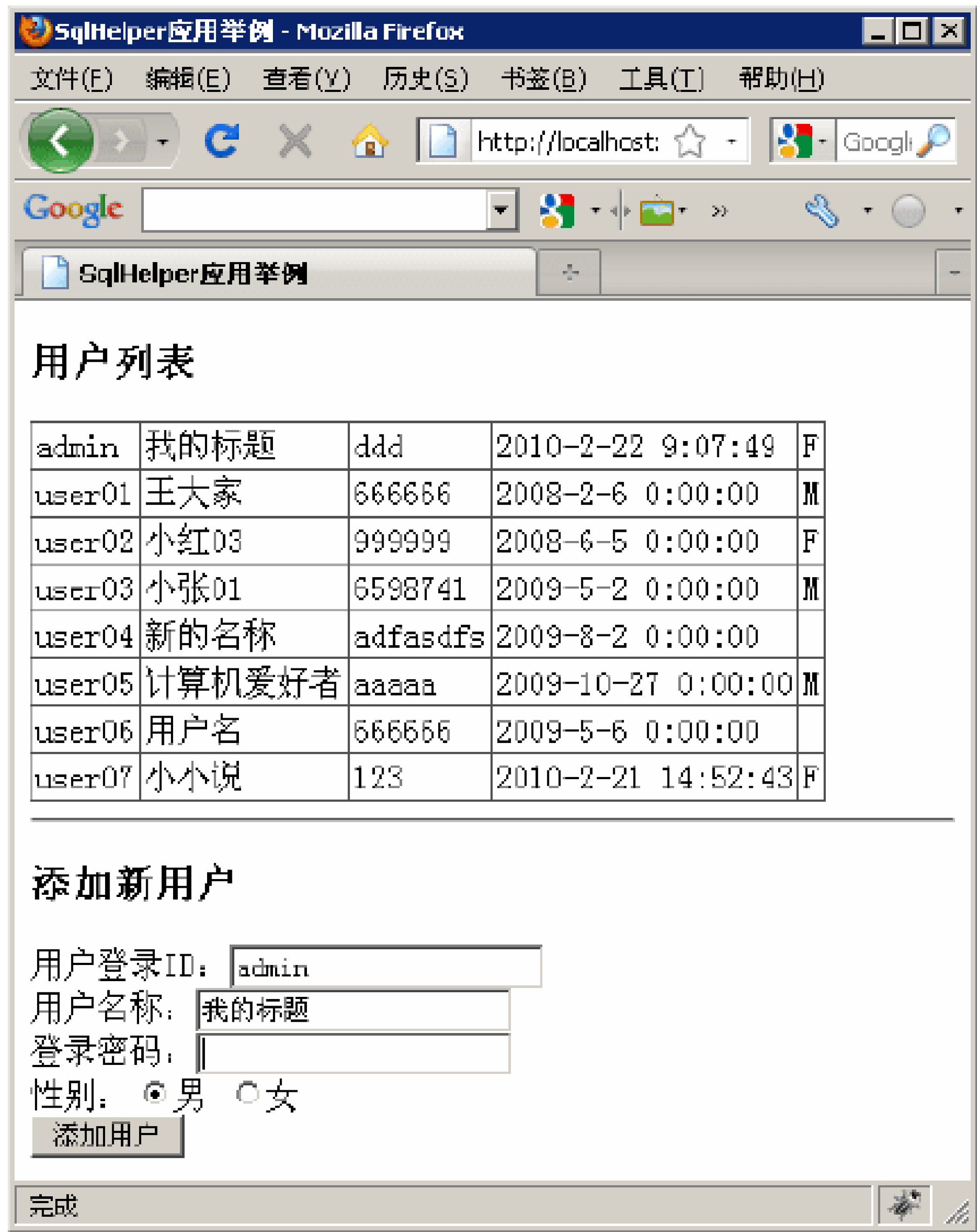


图 2.18　SqlHelper 应用示例

2.9　小　　结

本章介绍了.NET 数据访问框架 ADO.NET 的相关知识，包含 ADO.NET 组成、ADO.NET 包含的主要类、使用 ADO.NET 访问数据库的方法。使用原始的 ADO.NET 类和方法直接操作数据库是很烦琐的一个过程，本章最后还介绍了一个通用数据访问类 SqlHelper，封装了常用 ADO.NET 操作。使用 SqlHelper 类可以显著减少重复代码。

第 3 章　ASP.NET 数据控件

相对于其他 Web 开发技术来说，ASP.NET 的优势之一就是简单高效，能够用更少的代码完成同样的任务。数据库操作是大多数应用程序的一个核心功能，在代码工作量上占据很大比重。ASP.NET 对数据库操作进行了封装和简化，提供了一系列数据控件供开发人员使用。ASP.NET 数据控件可以分为两大类：数据源控件和数据绑定控件。数据源控件是管理连接到数据源及读取和写入数据等任务的 ASP.NET 控件。数据绑定控件是用来在页面上显示数据的控件。本章将介绍 ASP.NET 中的主要数据控件。

3.1　ASP.NET 数据绑定控件概述

数据绑定 Web 服务器控件是指可绑定到数据源控件，以实现在 Web 应用程序中轻松显示和修改数据的控件。使用数据绑定控件，不仅能够将控件绑定到一个数据结果集，还能够使用模板自定义控件的布局。同时，数据绑定控件还提供用于处理和取消事件的方便模型。

3.1.1　ASP.NET 主要数据绑定控件

数据绑定控件是用来在页面上显示数据的控件。数据绑定控件所显示的数据可以来自数据源控件，也可以来自一个集合对象，如 DataTable、DataSet，也可以来自 Array 或 List。将数据添加到数据绑定控件以进行显示的过程称为数据绑定。数据绑定控件功能强大，最基本的功能就是显示数据，还可以编辑数据，也能够按照指定的模板生成复杂的界面。ASP.NET 主要包括以下数据绑定控件。

（1）DataList：可用于显示任何重复结构中的数据，可按不同的布局显示行，例如按列或行对数据进行排序。

（2）DetailsView：可以逐一显示、编辑、插入或删除其关联数据源中的记录。即使 DetailsView 控件的数据源公开了多条记录，该控件每次也只会显示一条数据记录。

（3）FormView：可以处理数据源中的单条记录，该控件与 DetailsView 控件相似，二者区别在于在于 DetailsView 控件使用表格布局，而 FormView 控件则不指定用于显示记录的预定义布局。

（4）GridView：以表格的形式显示数据源中的值，该表格中的每一列代表一个字段，每一行代表一条记录。使用 GridView 控件可以选择和编辑这些项，也可以进行排序。

（5）Repeater：是一个数据绑定容器控件，用于生成各个项的列表。可使用模板定义网页上各个项的布局。当网页运行时，该控件为数据源中的每一项重复相应的布局。

3.1.2　最简单的数据绑定控件 DropDownList

在 ASP.NET 控件分类中，通常不把 DropDownList 分类为数据绑定控件，而是分类为标准控件。但是，DropDownList 控件确实具有数据绑定功能，从这个角度来看，和其他的数据绑定控件并无区别。由于 DropDownList 控件的简单性，很适合作为介绍数据绑定控件的每一个例子。

DropDownList 控件可以绑定到一个数据源，把数据源中的数据显示在列表中。在实际项目中，经常用到 DropDownList 控件显示数据库中的数据。例如，在很多页面上都可以看到让用户从下拉列表框中选择省份、选择职业等，如果这个页面是 ASP.NET 页面，那么这个控件通常就是一个 DropDownList，绑定到数据库中某个表。

用编程方式将数据绑定控件绑定到数据源，通常需要两个步骤。

（1）将数据绑定控件的 DataSource 属性设置为数据源（如一个 DataTable 的实例）。

（2）调用数据绑定控件的 DataBind()方法，执行数据绑定。

数据绑定控件都是直接或者间接继承自 BaseDataBoundControl 类，DataSource 属性和 DataBind 方法都在 BaseDataBoundControl 类中定义，签名如下。

```
public virtual Object DataSource { get; set; }
public override void DataBind()
```

将 DropDownList 控件绑定到数据源时，还需要设置另外两个属性：DataValueField 和 DataTextField。DataValueField 属性是指列表项中存储的值来自于数据库哪个字段，DataTextField 是指列表项中所显示的文本来自于数据库哪个字段。

【例 3-1】 DropDownList 显示数据。

本例从数据库中读取职业列表，并在 DropDownList 控件中显示。

提示： 为减少数据访问代码，节约篇幅，突出重点，本例使用了第 2 章所创建的 SqlHelper 类实现数据访问。本章后面的例子中也将使用 SqlHelper 类，不再一一说明。

（1）创建一个 ASP.NET Web 应用程序。

（2）在 App_Data 文件夹下添加一个数据库，在数据库中添加一个职业表 Occupation，创建表的 SQL 语句如下：

```
create table Occupation (OccupationID int identity(1,1) primary key,
OccupationName nvarchar(20) not null)
```

（3）在 Occupation 表中添加几条测试数据。

（4）在项目中添加一个页面 OccupationList.aspx，在页面上放置一个 DropDownList 控件，并且设置控件的 DataValueField 属性为 OccupationID（对应于数据库 Occupation 表的字段名称），设置控件的 DataTextField 属性为 OccupationName。在页面中添加 JavaScript 代码，当用户选择的职业发生改变时，弹出提示框。页面代码如下：

```
<head runat="server">
   <!--下面这段 Javascript 代码的功能为，当所选择的职业发生变化时，弹出提示框-->
   <script type="text/javascript" language="javascript">
      function showSelection() {
```

```
            var select = document.getElementById('list');
            if (select.selectedIndex > 0) {
                var option = select.options[select.selectedIndex];
                var msg = '你选择的职业是：' + option.text + '，职业代码为：' +
                option.value;
                alert(msg);
            }
        }
    </script>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    请选择一个职业：
        <asp:DropDownList ID="list" runat="server" DataTextField=
        "OccupationName" DataValueField="OccupationID" onchange=
        "showSelection();">
        </asp:DropDownList><br />
    </div>
    </form>
</body>
```

（5）在页面的 Page_Load 事件中，编写代码将数据绑定到 DropDownList 控件。代码如下：

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindOccupationList();
    }
}
private void bindOccupationList()
{
    SqlHelper db=new SqlHelper();           //SqlHelper 为通用数据访问类
    //创建命令以执行 select 语句检索数据
    var command = db.GetSqlStringCommond("select * from occupation");
    DataTable table = db.ExecuteDataTable(command);
    //将数据绑定到 dropdownlist 控件
    list.DataSource = table;
    list.DataBind();
}
```

（6）运行 OccupationList.aspx 页面，运行界面如图 3.1 所示。

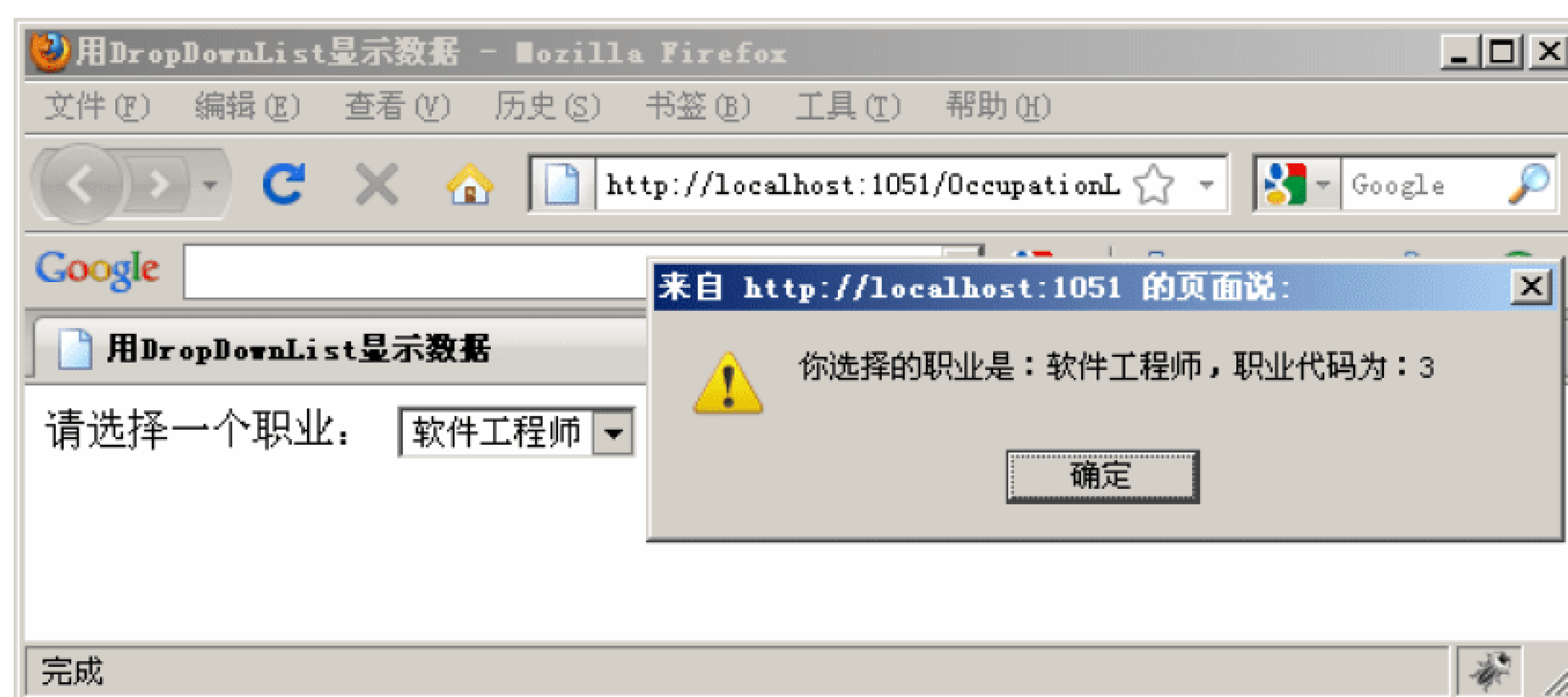


图 3.1　用 DropDownList 绑定数据

3.2　GridView 控件

GridView 是所有 ASP.NET 数据绑定控件中功能最强大的一个。功能的强大通常与性能的开销是成正比的，GridView 控件的性能相对来说也要弱一些。GridView 控件有非常复杂的属性、方法、事件列表，有许多使用技巧。本节将介绍 GridView 控件的使用。

3.2.1　显示数据

数据绑定控件最基本的功能就是显示数据。GridView 控件默认情况下以二维表格的形式显示数据。用户可以自定义表格中各个细节的样式，包括表头、表脚、每一列、奇偶行等，从而设计出美观的用户界面。下面先通过一个例子来看最基本的 GridView 控件的使用。

【例 3-2】 简单 GridView 示例。

本例演示 GridView 控件的基本使用，包括以表格形式显示数据、设置列标题、设置样式等。

（1）创建一个 ASP.NET 应用程序。

（2）在项目中添加一个数据库，在数据库中添加一个商品表 Products，表结构如下。

- ProductID：nchar(4)类型，商品编号，主键。
- ProductName：nvarchar(20)类型，不为空，商品名称。
- Price：float 类型，不为空，商品价格。
- Stock：float 类型，不为空，商品数量。
- Description：nvarchar(50)类型，可空，商品描述。

（3）向表中添加一些测试数据。

（4）在项目中添加一个新页面 SimpleGridView.aspx。在页面上放置一个 GridView 控件，代码如下：

```
<asp:GridView  ID="GridView1"  runat="server"  HorizontalAlign="Center"
CellPadding="2">
</asp:GridView>
```

（5）在页面的 Page_Load 事件中编写代码将数据绑定到 GridView 中。代码如下：

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindGrid();
    }
}
private void bindGrid()
{
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond("select * from products");
    DataTable table = db.ExecuteDataTable(command);
    GridView1.DataSource = table;
    GridView1.DataBind();
}
```

（6）运行 SimpleGridView.aspx 页面，运行效果如图 3.2 所示。

（7）在图 3.2 中，数据的列标题为数据库的字段名，界面不够友好。实际应用中，要使用直观的中文词语作为列标题。GridView 控件允许用户自定义列标题。在 ASP.NET 页面设计视图中，单击 GridView 控件右上角大于号按钮，则弹出 GridView 智能任务面板，如图 3.3 所示。

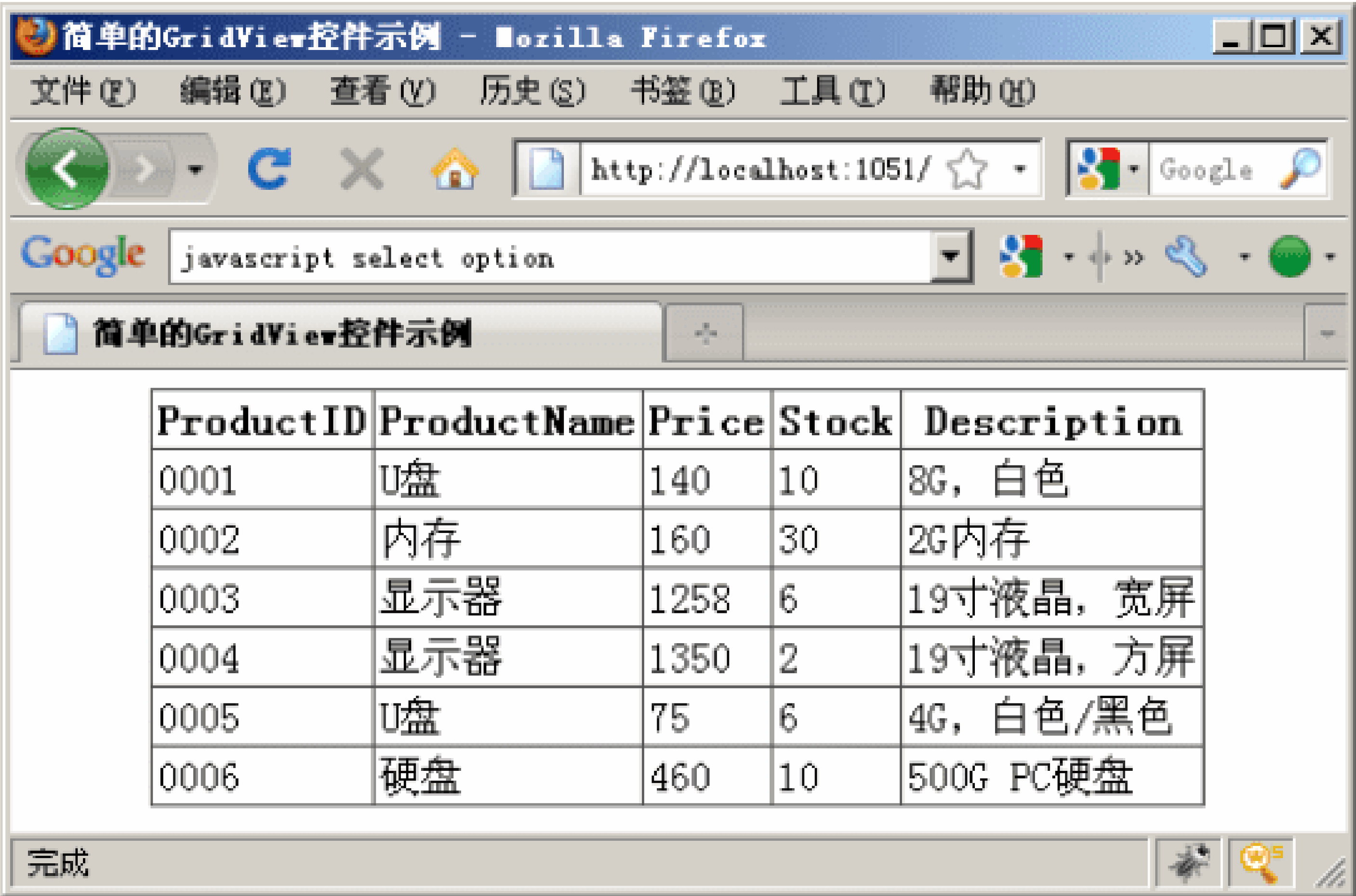


图 3.2　GridView 以表格形式显示数据

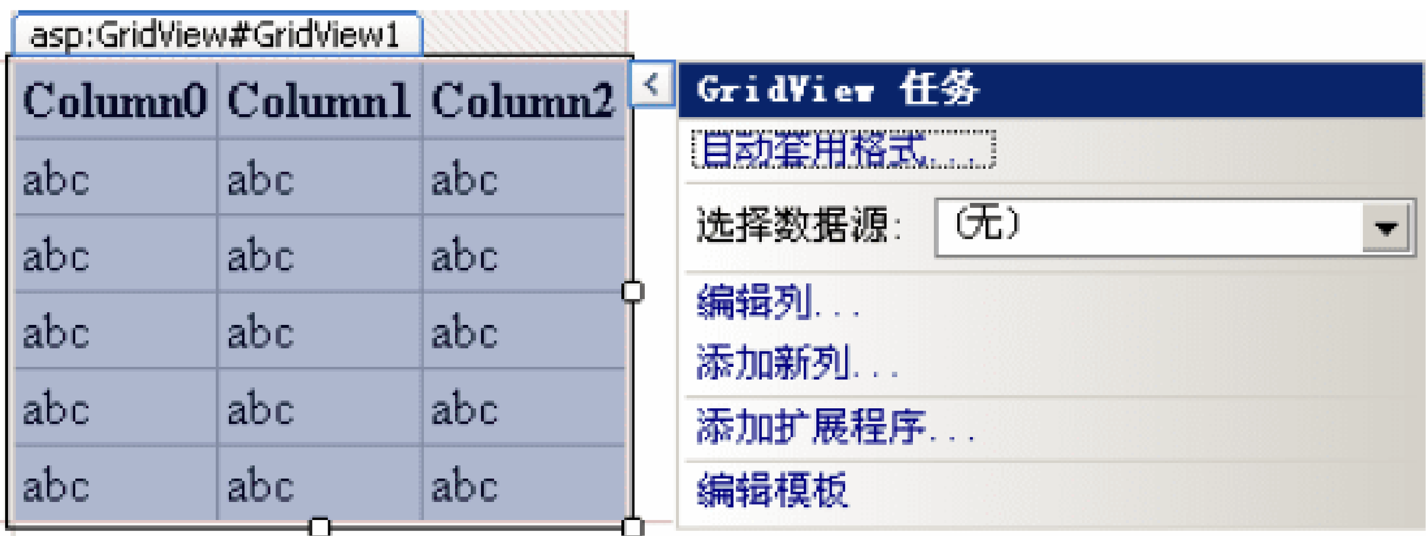


图 3.3　GridView 智能任务面板

在图 3.3 所示的 GridView 智能任务面板中，选择“编辑列”选项，则弹出如图 3.4 所示“字段”对话框。

在图 3.4 所示对话框中，“自动生成字段”复选框表示是否根据数据源自动生成列。由于接下来要定制列，不需要自动生成列，所以取消选中这个复选框。然后，从“可用字段”列表中选中“BoundField”项，单击“添加”按钮，则在对话框左下方的“选定的字段”列表中会添加一个新的字段，在对话框右边会显示这个字段的属性，如图 3.5 所示。

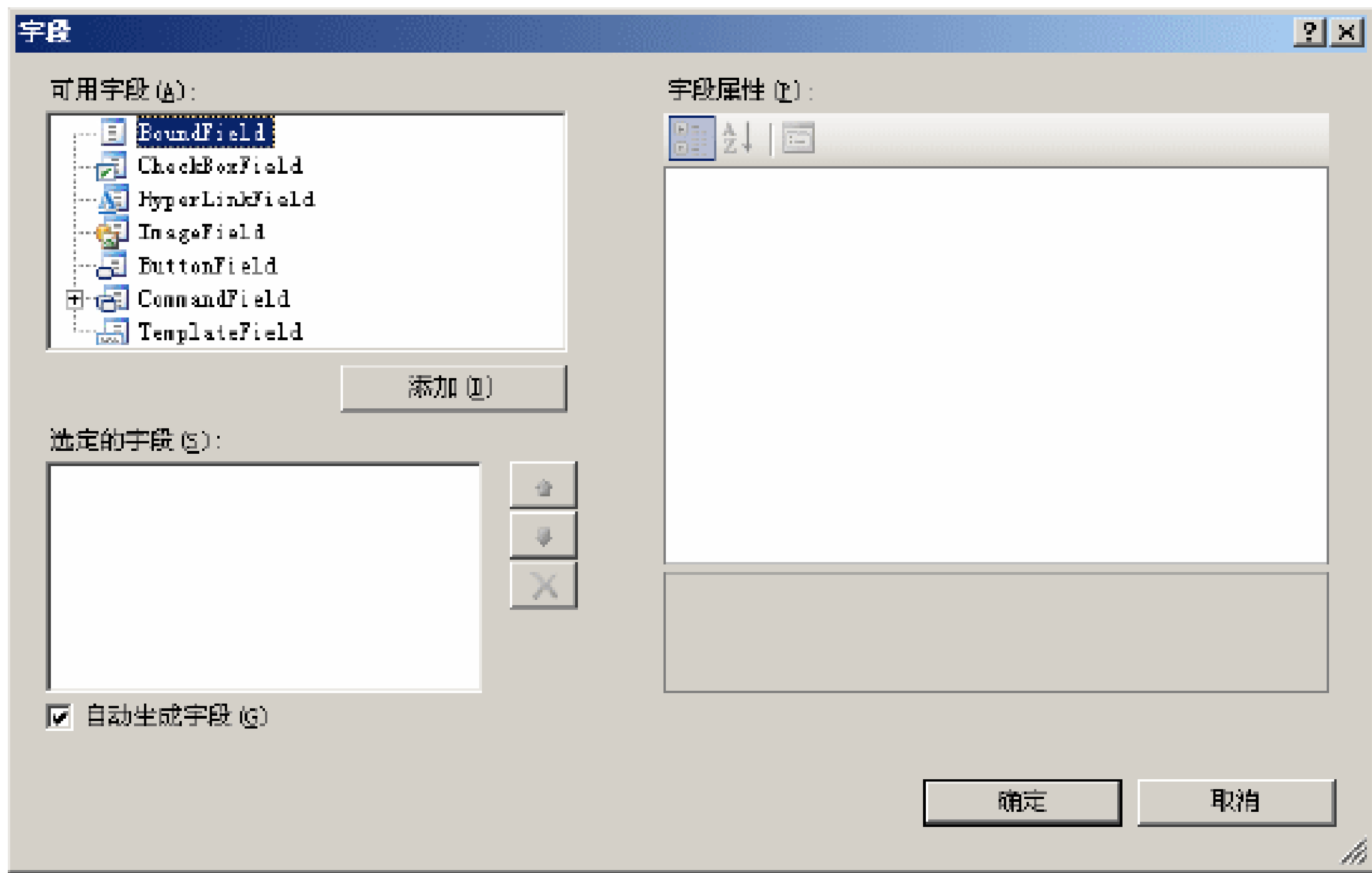


图 3.4　GridView 编辑列对话框

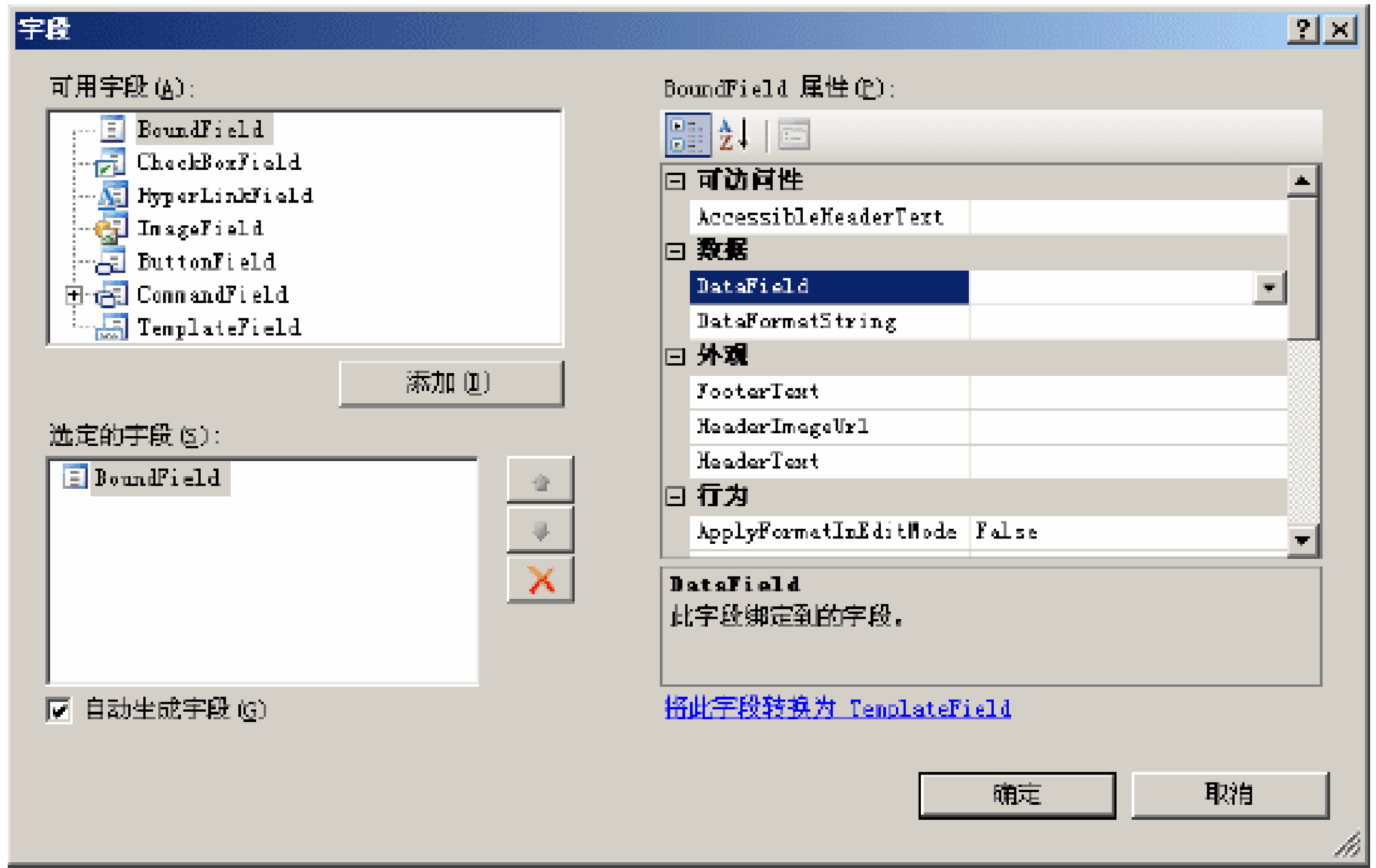


图 3.5　字段属性

在图 3.5 所示对话框的右侧属性面板中，DataField 属性表示要显示的数据库字段名，设置为 ProductID，HeaderText 属性表示列标题，设置为“商品编号”。到此为止为 GridView 添加了一列。按照同样的步骤，继续添加其他各列，包括 ProductName、Price、Stock、Description。注意在添加 Price 时，设置字段的 DataFormatString 属性为“{0:C}”，这个属性表示字段显示格式，此处以货币格式显示商品价格。

上述在设计视图中的操作会转换为对页面对应的 aspx 文件的修改。如果不采用设计视图，而是直接修改代码文件，也是完全可以的。前述编辑字段的操作对应的 GridView 代码

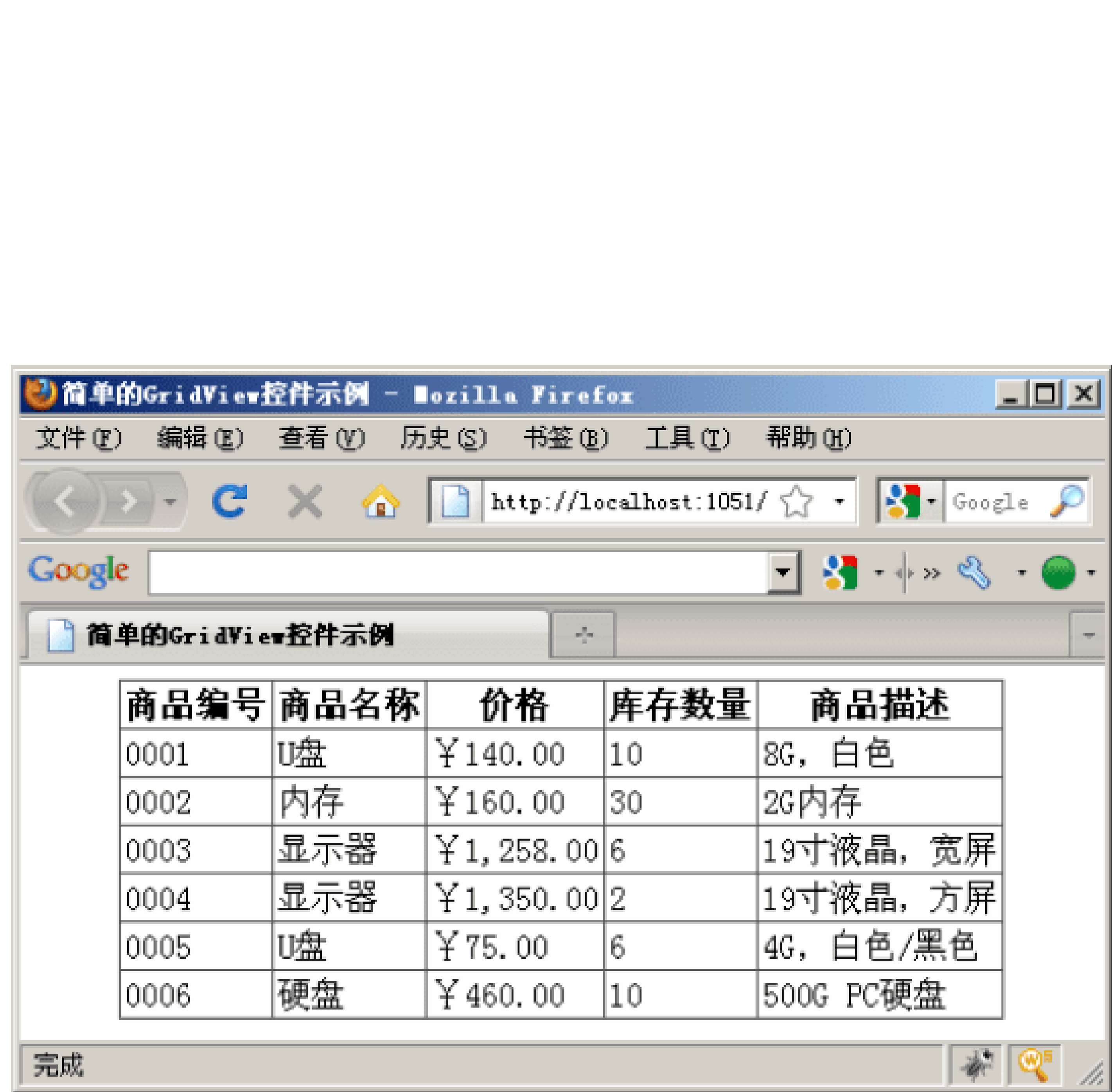
如下：

```
<asp:GridView ID="GridView1" runat="server" HorizontalAlign="Center"
CellPadding="2" AutoGenerateColumns="False">
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="商品编号" />
       <asp:BoundField DataField="ProductName" HeaderText="商品名称" />
        <asp:BoundField DataField="Price" DataFormatString="{0:C}" Header-
        Text="价格" />
        <asp:BoundField DataField="Stock" HeaderText="库存数量" />
        <asp:BoundField DataField="Description" HeaderText="商品描述" />
     </Columns>
</asp:GridView>
```

上述代码中，<asp:BoundField />表示这是一个数据绑定列，DataField 属性表示此列所绑定到的字段名，HeaderText 属性表示该列的标题，DataFormatString 属性表示显示列时使用的格式字符串。

（8）再次运行 SimpleGridView.aspx 页面，可以看到，列标题都变成了所指定的中文，商品价格也变成了货币，运行效果如图 3.6 所示。

（9）为了获得更加美观的页面，可以使用合理搭配的彩色显示 GridView 中的数据。GridView 允许定义各个行或列的前景色、背景色、字体、边框等样式。同时，GridView 还自带了几种经典样式以供用户选择。在图 3.3 所示的 GridView 智能任务面板中，选择“自动套用格式”命令，即可弹出如图 3.7 所示的“自动套用格式”对话框。从对话框左侧选择一种样式，从右侧即可预览此样式外观。单击“确定”按钮即可将选中样式应用于 GridView 控件。



商品编号	商品名称	价格	库存数量	商品描述
0001	U盘	￥140.00	10	8G，白色
0002	内存	￥160.00	30	2G内存
0003	显示器	￥1,258.00	6	19寸液晶，宽屏
0004	显示器	￥1,350.00	2	19寸液晶，方屏
0005	U盘	￥75.00	6	4G，白色/黑色
0006	硬盘	￥460.00	10	500G PC硬盘



图 3.6　设置了列标题和格式的 GridView　　　　图 3.7　自动套用格式对话框

应用了“传统型”格式的 GridView 代码如下：

```
<asp:GridView ID="GridView1" runat="server" HorizontalAlign="Center"
    CellPadding="4" AutoGenerateColumns="False" ForeColor="#333333"
    GridLines="None">
    <Columns>
```

```
            <asp:BoundField DataField="ProductID" HeaderText="商品编号" />
            <asp:BoundField DataField="ProductName" HeaderText="商品名称" />
            <asp:BoundField DataField="Price" DataFormatString="{0:C}"
            HeaderText="价格" />
            <asp:BoundField DataField="Stock" HeaderText="库存数量" />
            <asp:BoundField DataField="Description" HeaderText="商品描述" />
        </Columns>
       <%--RowStyle 指定了行的样式--%>
       <RowStyle BackColor="#EFF3FB" />
        <%--FooterStyle 指定了页脚样式--%>
       <FooterStyle BackColor="#507CD1" Font-Bold="True" ForeColor="White" />
        <%--PagerStyle 指定了分页区域的样式--%>
       <PagerStyle BackColor="#2461BF" ForeColor="White" HorizontalAlign=
       "Center" />
       <%--SelectedRowStyle 指定了选中行的样式--%>
       <SelectedRowStyle BackColor="#D1DDF1" Font-Bold="True" ForeColor=
       "#333333" />
       <%--HeaderStyle 指定了列标题的样式--%>
       <HeaderStyle BackColor="#507CD1" Font-Bold="True" ForeColor="White" />
       <%--EditRowStyle 指定了正在被编辑的行的样式--%>
        <EditRowStyle BackColor="#2461BF" />
       <%--AlternatingRowStyle 指定了偶数行的样式--%>
        <AlternatingRowStyle BackColor="White" />
</asp:GridView>
```

3.2.2　数据排序

GridView 支持数据排序功能。默认情况下，排序功能未启用，可以通过设置 AllowSorting 属性为 true 启用排序。如果 GridView 启用了排序功能，可排序的列标题显示为超链接，单击某一列标题，即可实现将数据按照这一列排序，再单击一次该列标题，则切换排序方式：升序变为降序，降序变为升序。

【例 3-3】 GridView 排序。

GridView 支持排序功能。本例将演示如何实现这个功能。

（1）按照例 3-2 所述步骤创建一个页面 GridViewSort.aspx，也可以直接使用例 3-2 所创建的页面 SimpleGridView.aspx。

（2）将 GridView 控件的 AllowSorting 属性设置为 true。

（3）将 GridView 控件中各个字段的 SortExpression 属性设置为各自的字段名称。设置完成后的 GridView 代码如下：

```
<asp:GridView ID="GridView1" runat="server" HorizontalAlign="Center"
   AutoGenerateColumns="False" AllowSorting="True" onsorting=
   "GridView1_Sorting" >
<Columns>
   <asp:BoundField DataField="ProductID" HeaderText="商品编号" SortEx-
   pression="ProductID" />
   <asp:BoundField DataField="ProductName" HeaderText="商品名称" SortEx-
   pression="ProductName" />
   <asp:BoundField DataField="Price" DataFormatString="{0:C}" HeaderText
   ="价格" SortExpression="Price"/>
   <asp:BoundField DataField="Stock" HeaderText="库存数量" SortExpression
   ="Stock" />
```

```
    <asp:BoundField DataField="Description" HeaderText="商品描述"  />
</Columns>
</asp:GridView>
```

（4）在 GridView 的 Sorting 事件中，按照所选择的字段对数据进行排序，重新绑定 GridView。代码如下：

```
protected void GridView1_Sorting(object sender, GridViewSortEventArgs e)
{
    bindGrid(e.SortExpression,
e.SortDirection==SortDirection.Descending);
}
/// <summary>
/// 根据排序字段绑定数据到 GridView
/// </summary>
/// <param name="sortField">要排序的字段</param>
/// <param name="desc">是否降序</param>
private void bindGrid(string sortField,bool desc)
{
    SqlHelper db = new SqlHelper();
    string sql = "select * from products ";
    //如果排序字段不为空，则在 select 语句后面添加 order by 子句
    if (!string.IsNullOrEmpty(sortField))
    {
        sql += " order by " + sortField;
        //如果 desc 参数为真，则在 select 语句中 order by 列名 后面添加 desc 关键字
        if (desc)
            sql += " desc ";
    }
    //执行查询命令，得到 DataTable，将其绑定到 GridView
    DbCommand command = db.GetSqlStringCommond(sql);
    DataTable table = db.ExecuteDataTable(command);
    GridView1.DataSource = table;
    GridView1.DataBind();
}
```

（5）在 Page_Load 事件中，编写代码加载全部数据。代码如下：

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindGrid(null,false);                          //加载所有数据，不排序
    }
}
```

（6）运行页面，可以看到所有设置了 SortExpression 属性的列标题都变成了链接按钮，单击列标题可以按照此列进行排序。运行界面如图 3.8 所示。

图 3.8　GridView 排序示例

3.2.3　数据分页

当需要显示的数据很多时，如果把数据全部显示在页面中，则页面会变得很大而且混乱。一种改进的做法是将数据分页显示，每页显示固定数量（如 10

条数据），用户通过翻页查看其余数据。GridView 支持数据分页功能，可以通过设置 AllowPaging 属性为 true 启用分页功能。

【例 3-4】 GridView 分页。

本例演示 GridView 如何分页显示数据。

（1）创建一个 ASP.NET 应用程序，把前面所创建的包含 Products 表的数据库复制到 App_Data 目录下。

（2）在项目中添加一个页面 GridViewPaging.aspx。

（3）在页面上放置一个 GridView，设置其各个字段，并将 GridView 控件的 AllowPaging 属性设置为 true。

（4）在页面上放置两个 TextBox 和两个 Button，用以设置 GridView 页面大小和当前页面。所有控件放置完成后 GridViewPaging.aspx 页面代码如下：

```
<form id="form1" runat="server">
<div>
    每页记录数<asp:TextBox ID="size" runat="server"></asp:TextBox>
    <asp:Button ID="Button1" runat="server" Text="修改" onclick="Button1_
    Click" /><br />
    转到指定页<asp:TextBox ID="page" runat="server"></asp:TextBox>
    <asp:Button ID="Button2" runat="server" Text="查看" onclick="Button2_
    Click" /><br /><br />
            <asp:GridView ID="GridView1" runat="server" HorizontalAlign=
            "Center"
            AutoGenerateColumns="False" AllowPaging="True"  PageSize="5"
                onpageindexchanging="GridView1_PageIndexChanging"    >
        <Columns>
            <asp:BoundField DataField="ProductID" HeaderText="商品编号"   />
            <asp:BoundField DataField="ProductName" HeaderText="商品名称" />
            <asp:BoundField DataField="Price" DataFormatString="{0:C}"
            HeaderText="价格" />
            <asp:BoundField DataField="Stock" HeaderText="库存数量" />
            <asp:BoundField DataField="Description" HeaderText="商品描述" />
        </Columns>
    </asp:GridView>
</div>
</form>
```

（5）在 Page_Load 事件中，绑定数据。代码如下：

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindGrid();
    }
}
//查询商品信息并绑定到 GridView 控件
private void bindGrid()
{
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond("select * from Products");
    DataTable table = db.ExecuteDataTable(command);
    GridView1.DataSource = table;
    GridView1.DataBind();
}
```

（6）在 GridView 的 PageIndexChanging 事件中，设置新的页号，并重新绑定数据。

```
protected    void    GridView1_PageIndexChanging(object    sender,
GridViewPageEventArgs e)
{
    GridView1.PageIndex = e.NewPageIndex;
    bindGrid();
}
```

（7）在设置页面大小按钮的 Click 事件中，编写代码修改 GridView 页面大小，并重新绑定数据。

```
protected void Button1_Click(object sender, EventArgs e)
{
    int n;
    if (int.TryParse(size.Text, out n))
    {
        GridView1.PageSize = n;
        bindGrid();
    }
}
```

（8）在跳转页面按钮 Click 事件中，编写代码设置 GridView 控件的当前页面索引，并重新绑定数据。

```
protected void Button2 Click(object sender, EventArgs e)
{
    int n;
    if(int.TryParse(page.Text,out n))
    {
        //GridView 的页号从 0 开始，而 TextBox 中输入的页号从 1 开始，所以绑定时要减 1
        GridView1.PageIndex = n-1;
        bindGrid();
    }
}
```

（9）运行页面，运行界面如图 3.9 所示。

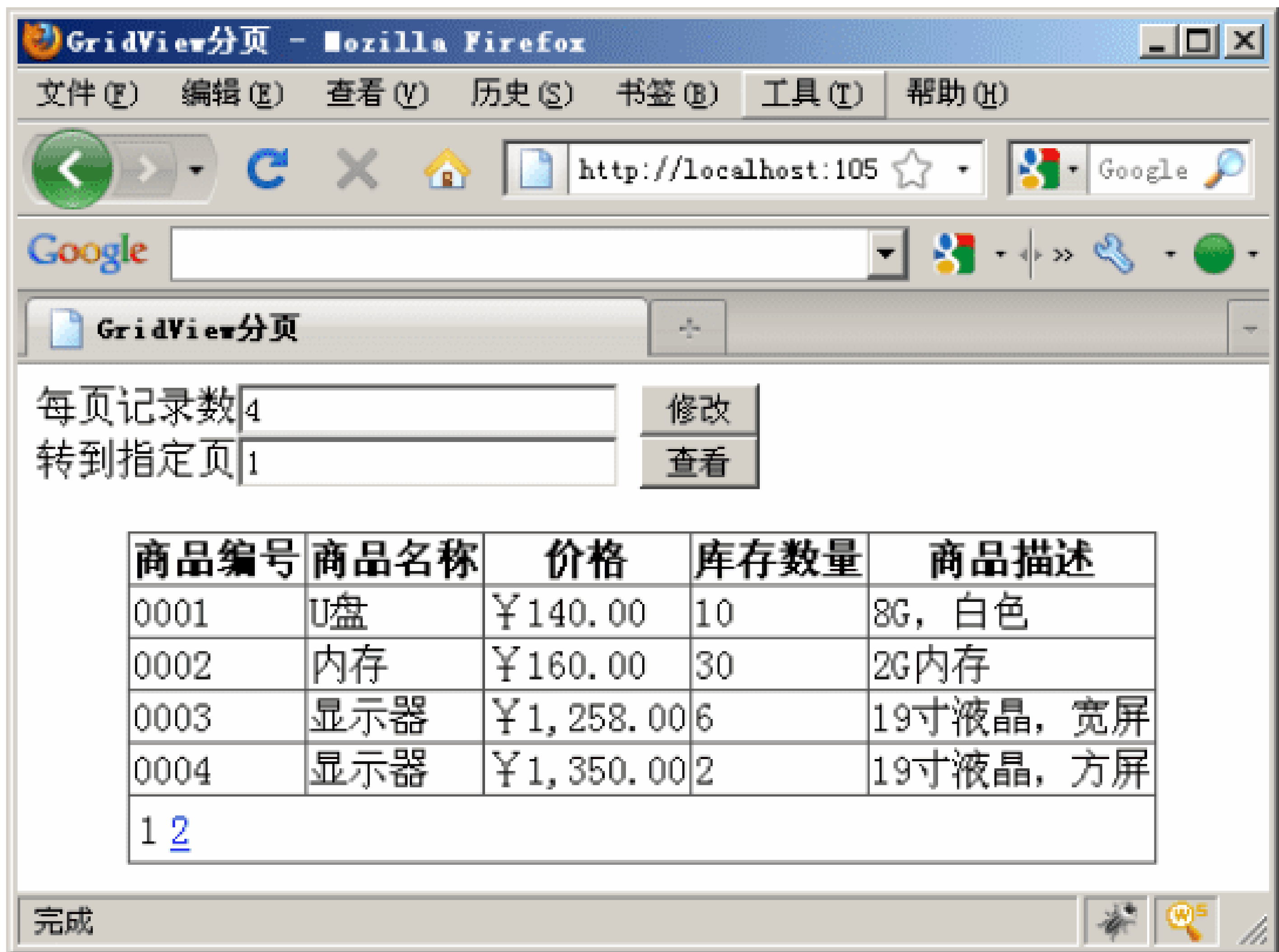


图 3.9　GridView 分页示例

例 3-4 利用了 GridView 自带的分页功能分页显示数据。这种方式虽然简单，却有非常严重的性能问题。通过例子中的 bindGrid 代码可以看出，程序首先把数据库中的所有数据读出，保存到 DataTable 中，然后再把 DataTable 绑定到 GridView 控件上，由 GridView 进行分页显示。如果数据库里数据量很大，例如一个网上商店可能会有上百万甚至更多商品，

那么把这些数据全部读出将浪费极大的服务器数据库和内存资源，大大降低服务器性能。

实际上，由于用户是一页一页查看数据的，程序没有必要把全部数据都从数据库读出，而只需要读出当前页需要显示的那部分数据。这种一页一页从数据库读取数据并且显示的分页方式才是正确的高性能分页方式。用这种方式进行数据分页时，通常的做法是写一个存储过程读取指定页面的数据。

综上所述，虽然 GridView 自带分页功能，但是这个功能在实际项目中应用很少，除非明确知道程序中的数据量非常少，否则不应该使用这种分页方式。正确的做法应该使用存储过程分页。下面通过一个例子具体说明存储过程分页的实现方法。

【例 3-5】 存储过程分页。

本例利用存储过程进行数据分页并显示在 GridView 中，例子中用到一个第三方控件 AspNetPager。

（1）创建一个 ASP.NET Web 应用程序，把例 3-4 所创建的数据库复制到项目中。

（2）创建分页存储过程，其功能就是根据页码和页面大小读取一写的数据。例如，页面大小为 10，当前要显示第 3 页，那么存储过程应该读取第 21 到 30 条数据。用一个公式来描述，要读取的数据为：从(pageIndex -1) * pageSize+1 到 pageIndex*pageSize。在 SQL Server 2005 中，有一个系统函数 Row_Number()可以得到一行数据在整个记录集中的行数。分页读取数据的存储过程代码如下：

```
CREATE PROCEDURE dbo.GetProductByPage
@pageIndex int,
@pageSize int
AS
    declare @min int;                                   //要读取数据的最小行号
    declare @max int;                                   //要读取数据的最大行号
    //计算@min 和@max
    set @min=@pageSize*(@pageIndex-1)+1;
    set @max=@pageSize*@pageIndex;
    //下面这条较为复杂的 sql 语句的作用是将小括号中的 select 语句结果当作一个表
       myTable
    //然后再从 myTable 中查询指定范围的数据
    with myTable as
    (
    select ProductID, ProductName, Price, Stock, Description,
    Row Number() over (order by ProductID) as rownum
    from Products
    )
    select ProductID, ProductName, Price, Stock, Description from myTable
    where rownum between @min and @max
    RETURN
```

提示： Row_Number()函数是 SQL Server 2005 中新增的函数，作用为获取当前行在查询结果集中的行数。Row_Number()函数后必须用 over 关键字指定查询的排序规则。

（3）在项目中添加一个页面 DbPaging.aspx，在页面上放置一个 GridView 以及 Label 和 TextBox，参照以下代码。

```
每页记录数<asp:TextBox ID="size" runat="server"></asp:TextBox>
<asp:Button ID="Button1" runat="server" Text="修改" onclick="Button1
Click" /><br />
```

```
转到指定页<asp:TextBox ID="pageNum" runat="server"></asp:TextBox>
<asp:Button ID="Button2" runat="server" Text="查看" onclick="Button2_
Click" /><br />
提示：共有<asp:Label ID="count" runat="server" Text=""></asp:Label>条数据
<br /><br /><br />
    <asp:GridView ID="GridView1" runat="server" HorizontalAlign="Center"
        AutoGenerateColumns="False"  >
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="商品编号"  />
        <asp:BoundField DataField="ProductName" HeaderText="商品名称" />
        <asp:BoundField DataField="Price" DataFormatString="{0:C}" Header
        Text="价格" />
        <asp:BoundField DataField="Stock" HeaderText="库存数量" />
        <asp:BoundField DataField="Description" HeaderText="商品描述" />
    </Columns>
</asp:GridView>
```

（4）从网上（http://down.chinaz.com/soft/10039.htm）下载第三方控件 AspNetPager，并添加到 Visual Studio 工具箱中。然后，把 AspNetPager 控件添加到 DbPaging.aspx 页面。

```
<webdiyer:AspNetPager ID="pager1" runat="server"  onpagechanged="pager1
PageChanged">
</webdiyer:AspNetPager>
```

（5）在 Page_Load 事件中，编写代码绑定数据。

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindGrid();
    }
}
//页面首次加载时调用此方法加载并显示全部数据
private void bindGrid()
{
    //查询所有商品信息并绑定到GridView控件
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond("select * from products");
    DataTable table = db.ExecuteDataTable(command);
    GridView1.DataSource = table;
    GridView1.DataBind();
    //读取商品总数并设置分页控件
    command = db.GetSqlStringCommond("select count(*) from products");
    int n = Convert.ToInt32(db.ExecuteScalar(command));
    count.Text = n.ToString();
    //将分页控件的总记录数和页面大小都设置为查询到的商品总数，从而在一页显示全部商品
    pager1.RecordCount = n;
    pager1.PageSize = n;
}
```

（6）在 AspNetPager 控件的 PageChanged 事件中，读取一页数据并显示在 GridView 中。

```
//读取一页数据并显示
private void bindPageData()
{
    SqlHelper db = new SqlHelper();
    //创建一个命令以存储过程
    DbCommand command = db.GetStoredProcCommond("GetProductByPage");
```

```
    //向存储过程添加两个参数（当前页号和页面大小）
    db.AddInParameter(command, "@pageIndex" ,DbType.Int32 , pager1.
    CurrentPageIndex);
    db.AddInParameter(command, "@pageSize", DbType.Int32, pager1.
    PageSize);
    //执行存储过程，得到数据，绑定到 GridView
    DataTable table = db.ExecuteDataTable(command);
    GridView1.DataSource = table;
    GridView1.DataBind();
}
//分页控件当前页发生变化时，重新绑定数据
protected void pager1 PageChanged(object sender, EventArgs e)
{
    bindPageData();
}
```

（7）在设置页面大小按钮的 Click 事件中，更改默认页面大小。

```
protected void Button1 Click(object sender, EventArgs e)
{
    int n;
    if (int.TryParse(size.Text, out n))
    {
        pager1.PageSize = n;
        bindPageData();
    }
}
```

（8）在跳转页面按钮的 Click 事件中，显示指定页面的数据。

```
protected void Button2 Click(object sender, EventArgs e)
{
    int n;
    if (int.TryParse(size.Text, out n))
    {
        pager1.CurrentPageIndex = n;
        bindPageData();
    }
}
```

（9）运行 DbPaging.aspx 页面，运行界面如图 3.10 所示。

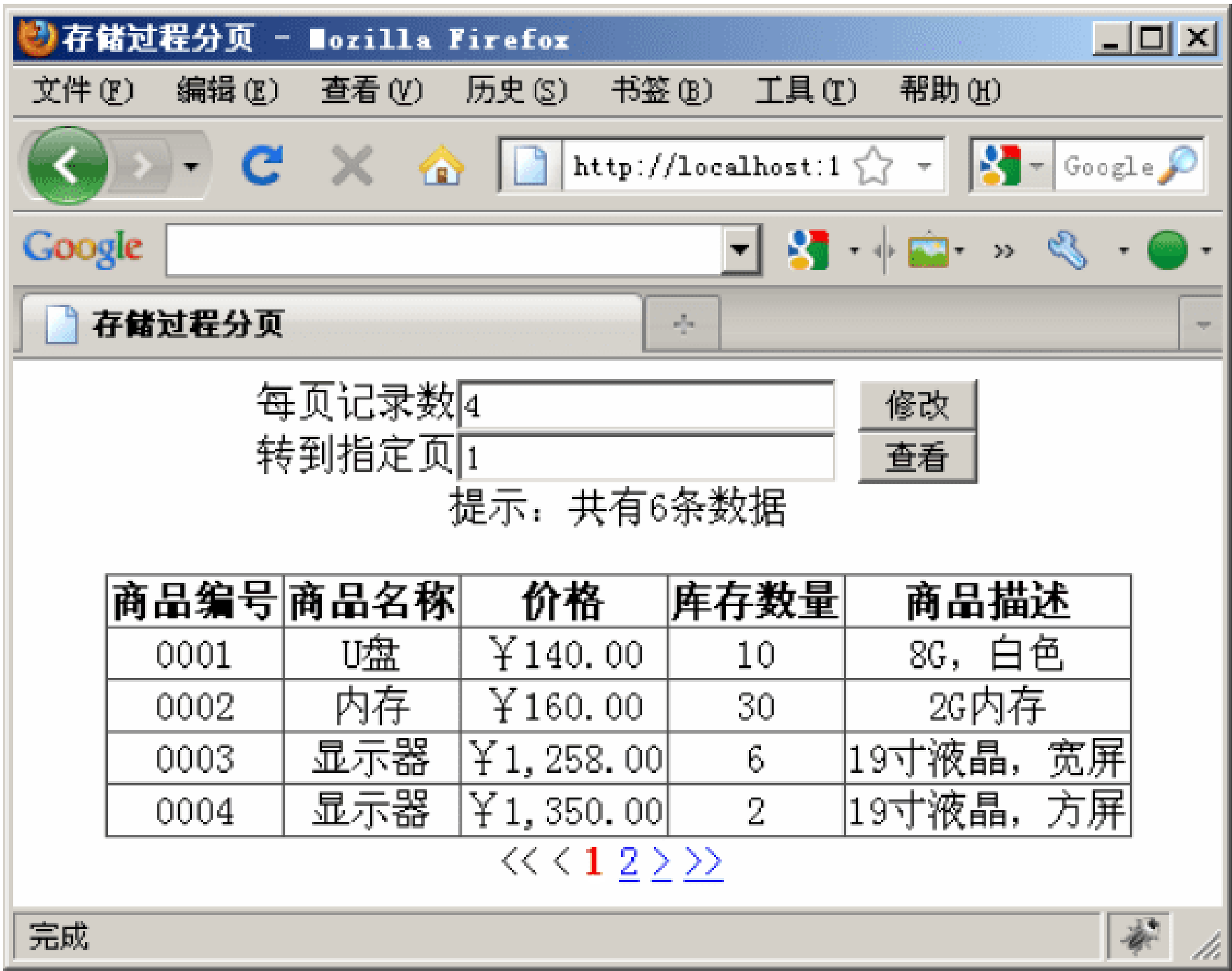


图 3.10　存储过程分页

3.2.4　删除数据

GridView 是一个功能强大的数据绑定控件，不仅能够显示数据，而且能够删除和编辑数据。当然，GridView 是一个数据绑定控件，仅提供数据显示和操作的界面，不执行对数据库的实际操作，如读取数据、更新数据等。用 GridView 编辑和删除数据时，必须配合代码或者数据绑定控件。前面所介绍的 GridView 的排序和分页功能也是需要代码或者数据源控件的配合才能工作。

【例 3-6】 删除数据。

本例演示如何在 GridView 中删除数据。

（1）创建一个 ASP.NET 应用程序，把前面所创建的包含 Products 表的数据库复制到 App_Data 目录下。

（2）在项目中添加一个页面 GridViewDelete.aspx。

（3）在页面上放置一个 GridView，设置其各个字段以显示 Products 表中商品信息。

```
<asp:GridView  ID="GridView1"  runat="server"  HorizontalAlign="Center"
AutoGenerateColumns="False">
   <Columns>
      <asp:BoundField DataField="ProductID" HeaderText="商品编号"   />
      <asp:BoundField DataField="ProductName" HeaderText="商品名称" />
      <asp:BoundField DataField="Price" DataFormatString="{0:C}" Header-
      Text="价格" />
      <asp:BoundField DataField="Stock" HeaderText="库存数量" />
      <asp:BoundField DataField="Description" HeaderText="商品描述" />
   </Columns>
</asp:GridView>
```

（4）在 GridViewDelete.aspx 页面的设计视图中，单击 GridView 控件右上角智能按钮打开智能面板，然后选择“添加新列”选项，则弹出如图 3.11 所示的“添加字段”对话框。

在图 3.11 所示“添加字段”对话框中，从顶部的“选择字段类型”下拉列表框中选择 CommandField，将页眉文本设置为“删除”，从“按钮类型”下拉列表框表中选择 Link，在命令按钮中选中“删除”。

提示：GridView 的字段有几种类型，最基本的是绑定字段 BoundField，其作用是以纯文本形式显示数据库里的数据。本例所使用的命令字段 CommandField 在 GridView 中显示为一个包含命令按钮的列，这些按钮可以触发服务器端的命令事件。GridView 的标准命令有 4 种：编辑 Edit、删除 Delete、更新 Update 和选择 Select。

经过如上操作以后，从页面代码视图中可以看到 GridView 代码多出一行。

```
<asp:CommandField HeaderText="删除" ShowDeleteButton="True" ShowHeader=
"True" />
```

（5）将 GridView 的 DataKeyNames 属性设置为 ProductID。DataKeyNames 属性表示 GridView 所显示数据的主键字段，GridView 的删除、编辑数据都需要根据主键生成相应的 SQL 语句。

（6）在 GridView 的 RowDeleting 事件中编写代码从数据库删除数据。

```
protected void GridView1_RowDeleting(object sender, GridViewDelete-
EventArgs e)
{
    string key = e.Keys[0].ToString();                    //得到要删除行的主键
    string sql = "delete from products where ProductID=@id";
                                                          //构建 delete 语句
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond(sql);
    db.AddInParameter(command, "@id", DbType.String, key); //添加参数
    db.ExecuteNonQuery(command);                          //执行删除
    bindGrid();                                           //重新绑定数据
}
```

（7）在页面的 Page_Load 事件中编写代码加载数据。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindGrid();
    }
}
private void bindGrid()
{
    //查询商品信息并绑定到 GridView
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond("select * from products");
    DataTable table = db.ExecuteDataTable(command);
    GridView1.DataKeyNames = new string []{"ProductID"};   //设置主键列名称
    GridView1.DataSource = table;
    GridView1.DataBind();
}
```

（8）运行 GridViewDelete.aspx 页面，运行效果如图 3.12 所示。

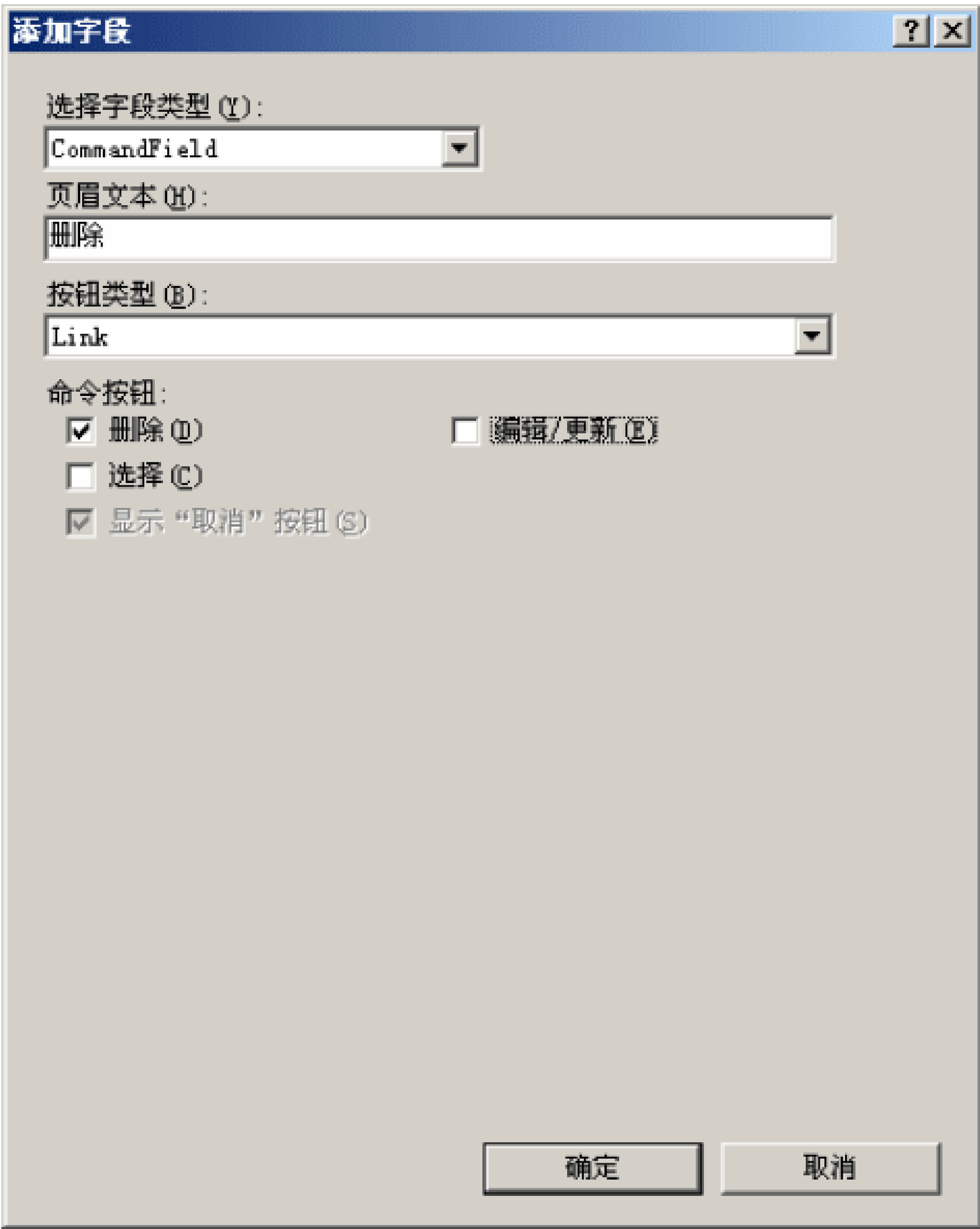


图 3.11　添加字段对话框



商品编号	商品名称	价格	库存数量	商品描述	删除
0001	U盘	￥140.00	10	8G，白色	删除
0002	内存	￥160.00	30	2G内存	删除
0003	显示器	￥1,258.00	6	19寸液晶，宽屏	删除
0004	显示器	￥1,350.00	2	19寸液晶，方屏	删除
0005	U盘	￥75.00	6	4G，白色/黑色	删除
0006	硬盘	￥460.00	10	500G PC硬盘	删除
0007	sfd	￥21.00	12	saggdsf	删除

图 3.12　GridView 删除数据

3.2.5　更新数据

在前面的例子中，GridView 处于只读状态，数据是以静态文本的形式显示在页面上，用户不能编辑。可以把 GridView 某一行设置为编辑状态，那么这一行数据的各个字段就分别显示在可编辑的控件中（默认为 TextBox）。GridView 的 EditIndex 属性表示处于编辑状态的行下标（从 0 开始），如果设置为–1，则表示没有行处于编辑状态。

如前所述，GridView 仅仅是一个显示和编辑数据的界面，不与数据库发生联系。要想把编辑的数据保存到数据库中，还需要在适当的事件中编写代码，完成实际的数据库操作。GridView 控件与编辑更新数据相关的事件主要有 RowEditing、RowCanelingEdit 和 RowUpdating。

【例 3-7】 更新数据。

本例演示如何在 GridView 中编辑和保存数据。

（1）创建一个 ASP.NET 应用程序，把前面所创建的包含 Products 表的数据库复制到 App_Data 目录下。

（2）在项目中添加一个页面 GridViewUpdate.aspx。

（3）在页面上放置一个 GridView，设置其各个字段以显示 Products 表中的商品信息。注意要把 ProductID 这一列的 ReadOnly 属性设置为 true，从而不允许用户编辑这一列。

```
<asp:GridView ID="GridView1" runat="server" HorizontalAlign="Center"
    onrowcancelingedit="GridView1_RowCancelingEdit"
    onrowediting="GridView1_RowEditing"
    onrowupdating="GridView1_RowUpdating"  AutoGenerateColumns="False">
   <Columns>
      <asp:BoundField DataField="ProductID" HeaderText="商品编号" ReadOnly
      ="true" />
      <asp:BoundField DataField="ProductName" HeaderText="商品名称" />
      <asp:BoundField DataField="Price" DataFormatString="{0:C}" Header-
      Text="价格" />
      <asp:BoundField DataField="Stock" HeaderText="库存数量" />
      <asp:BoundField DataField="Description" HeaderText="商品描述" />
   </Columns>
</asp:GridView>
```

（4）在 GridView 中添加一个“编辑”的命令字段。

```
<asp:CommandField HeaderText="编辑" ShowHeader="True" ShowEditButton=
"True" EditText="编辑" UpdateText="保存" CancelText="取消" />
```

（5）在 Page_Load 事件中，加载并显示数据。

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
   {
      bindGrid();
   }
}
private void bindGrid()
{
   //查询商品信息并绑定到 GridView
```

```
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond("select * from products");
    DataTable table = db.ExecuteDataTable(command);
    GridView1.DataSource = table;
    GridView1.DataBind();
}
```

（6）当用户在 GridView 中单击一行的编辑按钮时，会触发 GridView 的 RowEditing 事件。在此事件中编写代码，根据事件参数设置处于编辑状态的行。

```
protected void GridView1_RowEditing(object sender, GridViewEditEventArgs
e)
{
    GridView1.EditIndex = e.NewEditIndex;                    //设置编辑行
    bindGrid();                                              //重新绑定数据
}
```

（7）当用户在 GridView 中单击编辑行的取消按钮时，会触发 GridView 的 RowCanceling-Edit 事件。在此事件中再编写代码，取消当前行的编辑状态。

```
protected void GridView1 RowCancelingEdit(object sender, GridViewCancel-
EditEventArgs e)
{
    GridView1.EditIndex = -1;                                //取消编辑行
    bindGrid();
}
```

（8）在用户在 GridView 中单击编辑行的保存按钮时，会触发 GridView 的 RowUpdating 事件。在此事件中再编写代码，把用户编辑的数据保存到数据库。

```
protected void GridView1 RowUpdating(object sender, GridViewUpdate-
EventArgs e)
{
    string key = e.Keys[0].ToString();                       //获得主键列
    string name = e.NewValues["ProductName"].ToString(); //获得编辑后商品名称
    string price = e.NewValues["Price"].ToString();      //获得编辑后商品价格
    string stock = e.NewValues["Stock"].ToString();      //获得编辑后商品库存
    string description = e.NewValues["Description"].ToString();
                                                         //获得编辑后商品描述
                                                         //构建 update 语句
    string sql = "update Products set ProductName=@name, Price=@price,"
        +" Stock=@stock, Description=@description "
        + " where ProductID=@id";
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond(sql);
    //向 update 命令添加各个参数
    db.AddInParameter(command, "@name", DbType.String, name);
    db.AddInParameter(command, "@price", DbType.Double, price);
    db.AddInParameter(command, "@stock", DbType.Double, stock);
    db.AddInParameter(command, "@description", DbType.String, description);
    db.AddInParameter(command, "@id", DbType.String, key);
    db.ExecuteNonQuery(command);
    GridView1.EditIndex = -1;                                //取消编辑状态
    bindGrid();                                              //重新绑定数据
}
```

（9）运行 GridViewUpdate.aspx 页面，运行效果如图 3.13 所示。

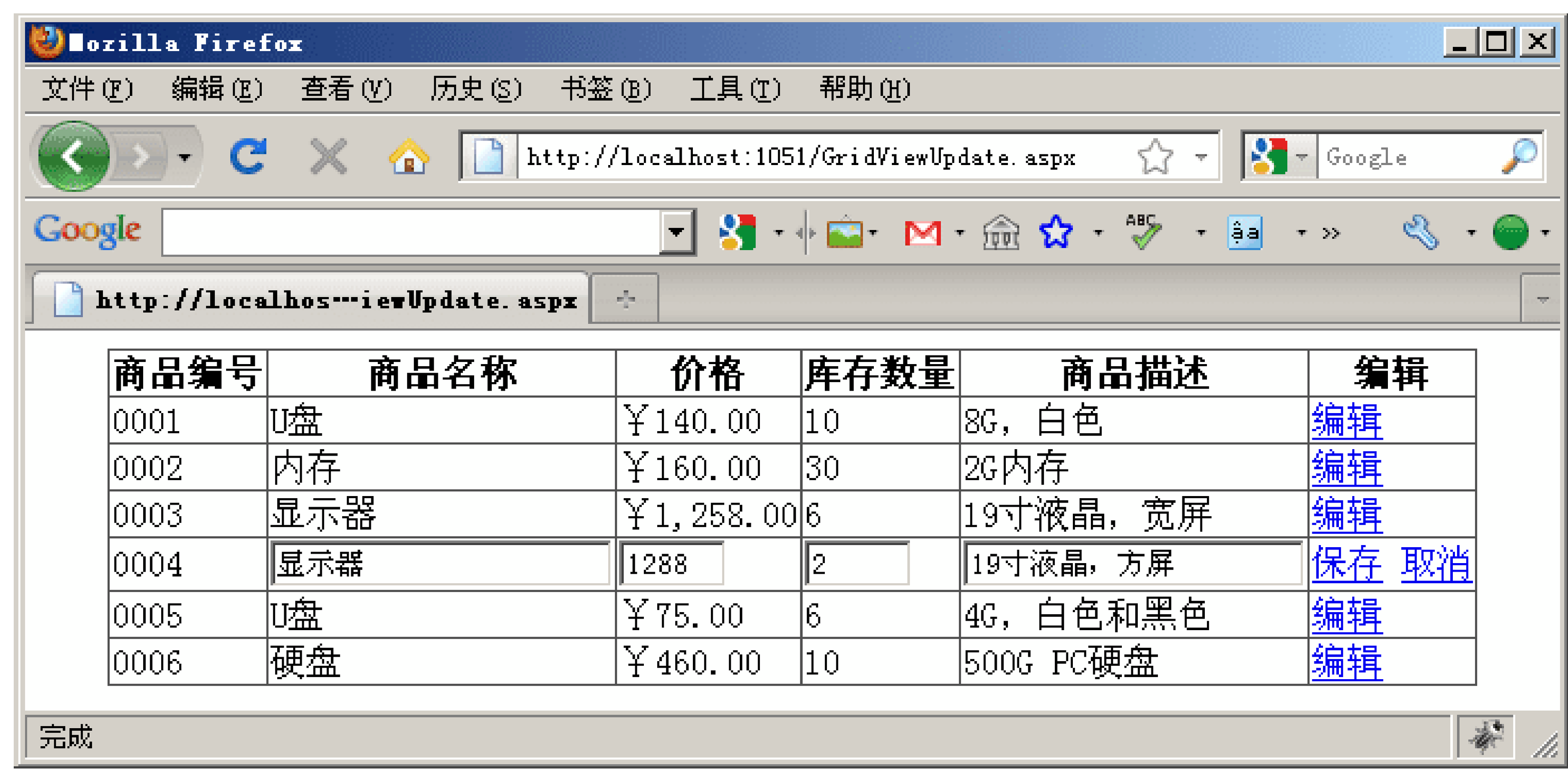


图 3.13　GridView 更新数据

3.2.6　光棒效果

在网页上经常看到这样一种特效，随着鼠标移动，鼠标下面的网页内容会高亮显示，这种效果为网页增色不少。在以表格形式显示数据的页面上，当鼠标在表格中移动时，鼠标移到的整个一行会高亮显示，鼠标移出时恢复正常显示，这种效果一般称为光棒。光棒效果是在浏览器端实现的，需要 JavaScript 支持。

【例 3-8】 GridView 实现光棒效果。

本例结合 GridView 控件和 JavaScript 脚本实现光棒效果。

（1）创建一个 ASP.NET 应用程序，把前面所创建的包含 Products 表的数据库复制到 App_Data 目录下。

（2）在项目中添加一个页面 LightBeam.aspx。

（3）在页面上放置一个 GridView，设置其各个字段以显示 Products 表中商品信息。为了使页面效果更加明显，设置 GridView 的表头、奇偶行背景色不一致。

```
<asp:GridView ID="GridView1" runat="server" HorizontalAlign="Center"
       AutoGenerateColumns="False" onrowcreated="GridView1_RowCreated"
      CellPadding="4" GridLines="None"  >
   <Columns>
      <asp:BoundField DataField="ProductID" HeaderText="商品编号"   />
      <asp:BoundField DataField="ProductName" HeaderText="商品名称" />
      <asp:BoundField DataField="Price" DataFormatString="{0:C}"
      HeaderText="价格" />
      <asp:BoundField DataField="Stock" HeaderText="库存数量" />
      <asp:BoundField DataField="Description" HeaderText="商品描述" />
   </Columns>
   <HeaderStyle BackColor="#0099FF" />
   <RowStyle BackColor="#EFF3FB" />
   <AlternatingRowStyle BackColor="White" />
</asp:GridView>
```

（4）在 Page_Load 事件中加载数据并绑定到 GridView。

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
   {
      //查询所有商品信息并绑定到GridView
```

```
        SqlHelper db = new SqlHelper();
        DbCommand command = db.GetSqlStringCommond("select * from
        products");
        DataTable table = db.ExecuteDataTable(command);
        GridView1.DataSource = table;
        GridView1.DataBind();
    }
}
```

（5）在 LightBeam.aspx 页面的 head 标记中编写两个 JavaScript 函数，实现鼠标进入和离开 GridView 某一行时改变和恢复背景色。鼠标进入时，把当前行的原有背景色保存到变量 originalColor 中，然后将当前行背景色设置为浅黄色；当鼠标离开时，将当前行背景色恢复成原来的背景色，即变量 originalColor 所保存的值。

```
<head runat="server">
    <title>GridView 光棒效果</title>
    <script type="text/javascript" language="javascript">
        //鼠标进行某行时调用此函数高亮显示该行
        //参数 e 表示鼠标移到的行
        function mouseenter(e) {
            originalColor = e.style.backgroundColor;     //得到原有背景色
            e.style.backgroundColor = '#ffffdd';         //设置新背景色
        }
        //鼠标移出某行时调用此函数恢复默认样式
        function mouseleave(e) {
            e.style.backgroundColor = originalColor;     //恢复成原来的背景色
        }
    </script>
</head>
```

（6）在 GridView 控件的 RowCreated 事件中，编写代码，将每一行的鼠标进入和离开事件关联到刚才所写的 JavaScript 函数。

```
protected void GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType != DataControlRowType.DataRow)
        return;
    //调用 mouseenter()和 mouseover()函数，把当前行作为参数传递
    e.Row.Attributes.Add("onmouseover", "mouseenter(this);");
    e.Row.Attributes.Add("onmouseout", "mouseleave(this);");
}
```

（7）运行 LightBeam.aspx 页面，运行效果如图 3.14 所示。



图 3.14　GridView 光棒效果

在例 3-8 中，为页面上的一个 GridView 控件实现了光棒效果。如果 ASP.NET 项目中有许多页面的 GridView 都需要实现光棒效果，按照例 3-8 的方法，就需要为每个页面和每个 GridView 都重复相同的工作：在页面上添加两个 JavaScript 函数，在 GridView 控件的 RowCreated 事件中为每一行的鼠标事件注册 JavaScript 函数。这种方法显然不够优秀。对于这个问题有一个更好的解决方案，就是编写一个实现了光棒效果的 GridView 控件，然后在需要的页面上直接使用这个控件即可，页面中不需要写任何代码。

【例 3-9】 自定义 GridView 控件。

本例设计了一个自定义 GridView 控件实现光棒效果。

（1）在 Visual Studio 中新建一个 ASP.NET 服务器控件项目 MyControls。

（2）在 MyControls 项目中添加一个类 MyGridView，此类从 GridView 中继承。

```
[ToolboxData("<{0}:MyGridView runat=\"server")]
public class MyGridView:GridView    {    }
```

（3）在 MyGridView 类中定义两个属性，分别表示鼠标划过时的背景色和前景色（文字颜色）。

```
public Color hoverBackColor { get; set; }
public Color hoverTextColor { get; set; }
```

（4）在 MyControls 项目中添加一个 JavaScript 文件 grid.js，在其中编写鼠标进入和移出的函数。

```
//鼠标移进某行时调用此函数，设置行的背景和前景色
//参数：row 当前行，bgColor 背景色，textColor 前景色
function MouseEnter(row, bgColor, textColor) {
    row.style.backgroundColor = bgColor;
    row.style.color = textColor;
}
//鼠标移出时调用此函数，恢复行的背景色和前景色
function MouseLeave(row) {
    row.style.backgroundColor                                =
row.getAttribute("OriginalColor");
    row.style.color                                          =
row.getAttribute("OriginalTextColor");
}
```

（5）在解决方案资源管理器中选中 grid.js 文件，在属性窗口中，将其生成操作属性设置为“嵌入的资源”，如图 3.15 所示。通过把 grid.js 设置为嵌入的资源，在生成项目时，会把 grid.js 也包含在生成的程序集中。

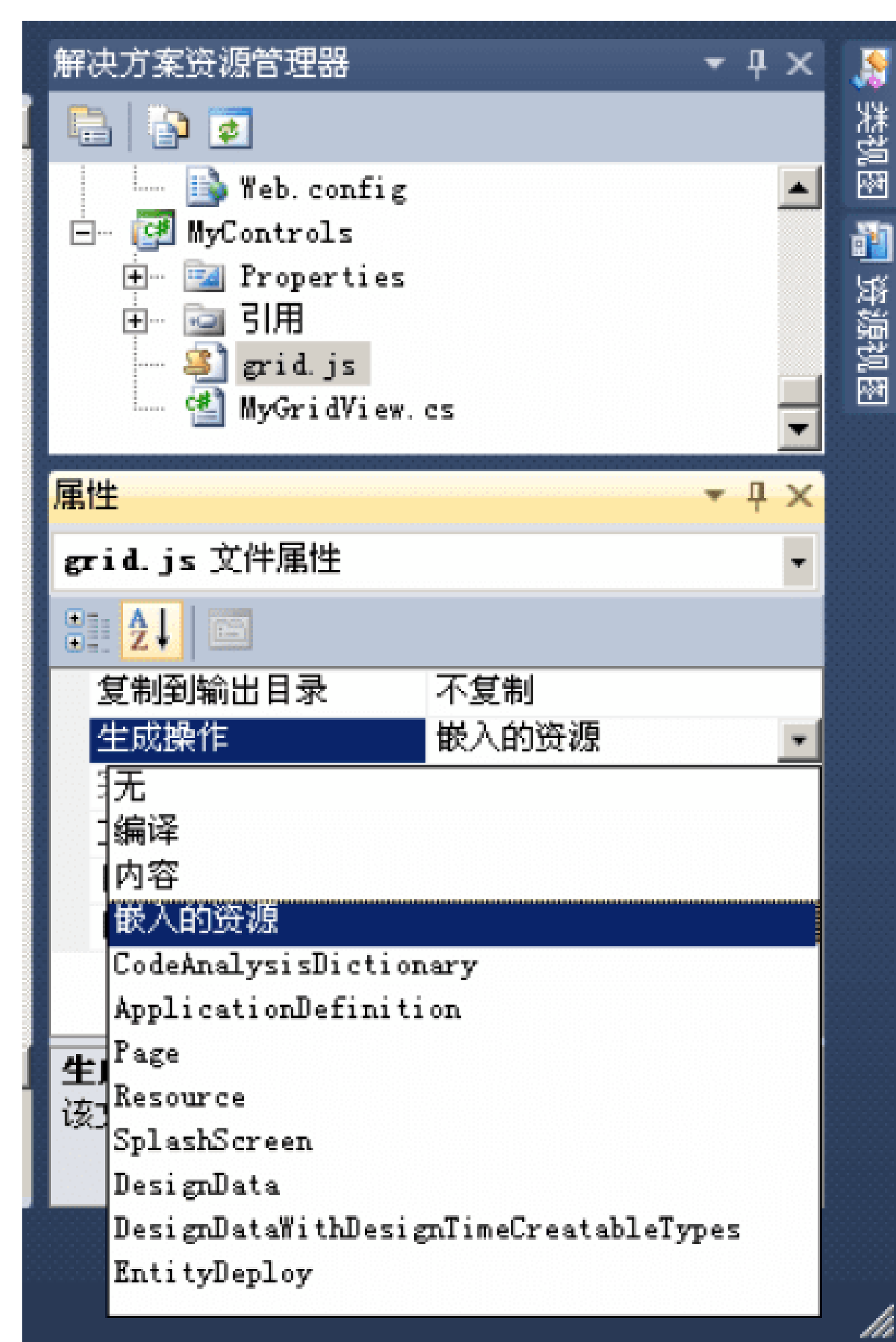


图 3.15　将文件设置为嵌入的资源

（6）在 MyControls 项目 Properties 文件夹下，打开 AssemblyInfo.cs 文件，在其中添加如下代码：

```
//注册 gird.js 为 javascript 资源
[assembly:System.Web.UI.WebResource("MyCon
trols.grid.js","text/javascript")]
```

（7）在 MyGridView 类中，重写 OnPreRender()方法，引用刚才所定义的 JavaScript 资源。

```
protected override void OnPreRender(EventArgs e)
{
    base.OnPreRender(e);
    string resourceName = "MyControls.grid.js";
    ClientScriptManager cs = this.Page.ClientScript;
    cs.RegisterClientScriptResource(typeof(MyControls.MyGridView),
    resourceName);
}
```

（8）在 MyGridView 类中，重写 OnRowCreated()方法，注册浏览器端鼠标移入和移出事件，调用相应的 JavaScript 函数。

```
protected override void OnRowCreated(GridViewRowEventArgs e)
{
    base.OnRowCreated(e);
    GridViewRow row = e.Row;
    //如果当前被创建的行不是数据行则不需要处理，函数返回
    if (row.RowType != DataControlRowType.DataRow)
        return;
    Color backColor = Color.Empty;
    Color textColor = Color.Empty;
    //根据是奇偶行得到原来的背景色和文字色
    if (row.RowIndex % 2 == 0)
    {
        backColor = (this.RowStyle.BackColor == Color.Empty) ? Color.White :
        this.RowStyle.BackColor;
        textColor = (this.RowStyle.ForeColor == Color.Empty) ? Color.Black :
        this.RowStyle.ForeColor;
    }
    else
    {
        backColor = (this.AlternatingRowStyle.BackColor == Color.Empty) ?
        Color.White : this.AlternatingRowStyle.BackColor;
        textColor = (this.AlternatingRowStyle.ForeColor == Color.Empty) ?
        Color.Black : this.AlternatingRowStyle.ForeColor;
    }
    //注册鼠标移动事件
    row.Attributes.Add("onmouseover", String.Format("MouseEnter(this,
    '{0}','{1}')",
        ColorTranslator.ToHtml(this.hoverBackColor),
        ColorTranslator.ToHtml(this.hoverTextColor)));
    row.Attributes.Add("onmouseout", "MouseLeave(this)");
    //将数据行原有颜色作为参数添添加到行中，以供 javascript 使用
    row.Attributes.Add("OriginalColor", ColorTranslator.ToHtml
    (backColor));
    row.Attributes.Add("OriginalTextColor", ColorTranslator.ToHtml
    (textColor));
}
```

（9）创建一个 ASP.NET 项目以测试 MyGridView 自定义控件。新建一个 ASP.NET 项目，在项目中添加一个页面 MyGridTestPage.aspx，在页面上放置 MyGridView 控件，设置控件的 hoverTextColor 和 hoverBackColor 属性。

提示: 如果 Visual Studio 工具箱中没有 MyGridView 自定义控件，则重新生成 MyControls 项目，MyGridView 控件就会出现在工具箱中。

```
<%@ Register assembly="MyControls" namespace="MyControls" tagprefix="cc2"
%>
...
```

```
<cc2:MyGridView ID="GridView1" runat="server" hoverBackColor="#ffffcc"
hoverTextColor="Red"
 AutoGenerateColumns="false" >
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="商品编号"  />
        <asp:BoundField DataField="ProductName" HeaderText="商品名称" />
        <asp:BoundField DataField="Price" DataFormatString="{0:C}" Header-
        Text="价格" />
        <asp:BoundField DataField="Stock" HeaderText="库存数量" />
        <asp:BoundField DataField="Description" HeaderText="商品描述" />
    </Columns>
      <RowStyle BackColor="#EFF3FB" />
      <AlternatingRowStyle BackColor="White" />
</cc2:MyGridView>
```

（10）在 Page_Load 事件中将数据绑定到 MyGridView。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
         SqlHelper db = new SqlHelper();
        DbCommand command = db.GetSqlStringCommond("select * from
        products");
        DataTable table = db.ExecuteDataTable(command);
        GridView1.DataSource = table;
        GridView1.DataBind();
    }
}
```

（11）运行页面，运行界面见图 3.16。

虽然例 3-9 和 3-8 运行界面看起来很相似，但二者实现思路完全不同。例 3-9 的自定义控件是可复用的，任何页面上只要放置一个 MyGridView 控件，不需要编写代码就可以实现光棒效果。另外，二者所产生的客户端 HTML 代码也不一样。



商品编号	商品名称	价格	库存数量	商品描述
0001	U盘	￥140.00	10	8G，白色
0002	内存	￥160.00	30	2G内存
0003	显示器	￥1,258.00	6	19寸液晶，宽屏
0004	显示器	￥1,288.00	2	19寸液晶，方屏
0005	U盘	￥75.00	6	4G，白色和黑色
0006	硬盘	￥460.00	10	500G PC硬盘

图 3.16　自定义 GridView 控件实现光棒效果

在 MyGridView 自定义控件中，注册了 grid.js 文件作为资源。在使用 MyGridView 的页面上，会自动添加一条引用 JavaScript 文件的语句。例如，MyGridTestPage.aspx 页面所生成的 HTML 代码中包含以下语句。

```
<script
src="/WebResource.axd?d=9zpWT3uCqAMxPp6-4mxKUXe1otMVa78tSIYrE443z7M1&am
p;t=634026979432812500" type="text/javascript"></script>
```

3.2.7　数据汇总

当用 GridView 显示报表性质的数据（如销售记录、考试成绩等）时，通常需要在表格底部显示汇总信息，如总销售量、总销售额、平均成绩等。另外，用户经常需要将 GridView 中的数据导出至 Excel，以方便后续使用。

【例 3-10】 数据汇总。

本例演示 GridView 的数据汇总和数据导出。

（1）创建一个 ASP.NET 应用程序，把前面所创建的包含 Products 表的数据库复制到 App_Data 目录下。

（2）在项目中添加一个页面 GridViewSummary.aspx。

（3）在页面上放置一个 GridView，设置其各个字段以显示 Products 表中的商品信息。由于要在 GridView 的 Footer 中显示汇总数据，所以将其 ShowFooter 属性设置为 true。在页面上放置一个 Button，用以导出数据到 Excel。

```
<asp:Button runat="server" Text="导出 Excel" id="button1"  onclick=
"button1_Click" /><br />
<asp:GridView ID="grid" runat="server" HorizontalAlign="Center"  AutoGene-
rateColumns="False" showFooter="true"
onrowdatabound="grid_RowDataBound" >
<Columns>
    <asp:BoundField DataField="ProductID" HeaderText="商品编号"   />
    <asp:BoundField DataField="ProductName" HeaderText="商品名称" />
    <asp:BoundField DataField="Price" HeaderText="价格" />
    <asp:BoundField DataField="Stock" HeaderText="库存数量" />
    <asp:BoundField DataField="Description" HeaderText="商品描述" />
</Columns>
<HeaderStyle BackColor="#0099FF" />
<FooterStyle BackColor="#EFF3FB" />
</asp:GridView>
```

（4）在 GridView 的 RowDataBound 事件中，逐行累加计算商品总金额，并在 GridView 的 Footer 中显示汇总数据。

```
private double totalStock=0;                                    //总库存
private double totalMoney=0;                                    //总金额
protected void grid_RowDataBound(object sender, GridViewRowEventArgs e)
{
    //如果当前行为数据行，逐行累加各行数据
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        double stock = Convert.ToDouble(e.Row.Cells[3].Text);
        double price = Convert.ToDouble(e.Row.Cells[2].Text);
        totalStock += stock;
        totalMoney += stock * price;
    }
    //如果当前行为 Footer，则在 Footer 中显示汇总数据
    else if (e.Row.RowType == DataControlRowType.Footer)
    {
        e.Row.Cells[0].Text = "商品总库存";
        e.Row.Cells[1].Text = totalStock.ToString();
        e.Row.Cells[2].Text = "商品总金额";
        e.Row.Cells[3].Text = totalMoney.ToString();
    }
}
```

（5）在“导出”按钮的 Click 事件中编写代码导出数据。

```
protected void button1_Click(object sender, EventArgs e)
{
    Response.Clear();
```

```
    Response.AddHeader("content-disposition", "attachment; filename=商品信
    息.xls");
    Response.Charset = "";
    Response.ContentType = "application/excel";        //内容类型为 Excel
    System.IO.StringWriter stringWrite = new System.IO.StringWriter();
    System.Web.UI.HtmlTextWriter htmlWrite = new HtmlTextWriter
    (stringWrite);
    grid.RenderControl(htmlWrite);
    Response.Write(stringWrite.ToString());
    Response.End();
}
```

（6）在 GridViewSummary.aspx 页面中重写 VerifyRenderingInServerForm()方法，取消对 GridView 控件的验证。否则程序运行时会出现以下错误提示“类型 GridView 的控件必须放在具有 runat=server 的窗体标记内”。

```
public override void VerifyRenderingInServerForm(Control control)
{
    if (control == grid) return;                      //不验证 GridView 控件
    base.VerifyRenderingInServerForm(control);
}
```

（7）运行 GridViewSummary.aspx 页面，运行效果如图 3.17 所示，导出的 Excel 文件如图 3.18 所示。

图 3.17　GridView 数据汇总和导出

图 3.18　导出的 Excel 数据

3.3　DataList 控件

DataList 控件是一种很常用的数据绑定控件，可以用自定义格式显示数据。DataList 控件绑定到一个数据源，数据源中每一条数据在 DataList 中称为一项。DataList 使用模板来定义数据显示格式。DataList 可以为项、交替项、选定项和编辑项创建模板，也可以使用标题、脚注和分隔符模板自定义 DataList 的整体外观。

3.3.1　以表格形式显示数据

DataList 控件最基本的功能是以纯文本的表格形式显示数据。这一功能与 GridView 类

似，但从开发人员角度来看，二者的实现方法不同。在 GridView 中显示数据时，需要向 GridView 中添加各个绑定字段，而用 DataList 显示数据时，需要定义项模板。

DataList 顾名思义，是一个数据列表，可以显示多条数据。每一条数据称为 DataList 的一项，如何在控件中显示一项称为项模板。例如，如果要显示性别，是用 Label 显示，还是用 TextBox 显示，还是用 RadioButton 显示，这就是项模板要解决的问题。当然，项模板的作用绝不仅是用不同类型的控件显示数据，还包括控件布局等强大功能。

【例 3-11】 DataList 以表格形式显示数据。

本例演示 DataList 以表格形式显示商品信息。要注意体会例子中项模板的定义和使用。

（1）新建一个 ASP.NET Web 应用程序，将前面所创建的包含 Products 表的数据库复制到项目中 App_Data 文件夹下。

（2）在页面上放置一个 DataList 控件。此时页面代码如下：

```
<asp:DataList ID="DataList1" runat="server">    </asp:DataList>
```

（3）在页面设计视图中，单击 DataList 控件右上角的智能按钮，打开智能任务面板，从中选择“编辑模板”选项，如图 3.18 所示。

选择了“编辑模板”选项以后，DataList 控件从图 3.19 所示的一个灰色固定只读变成了如图 3.20 所示的可编辑控件容器。

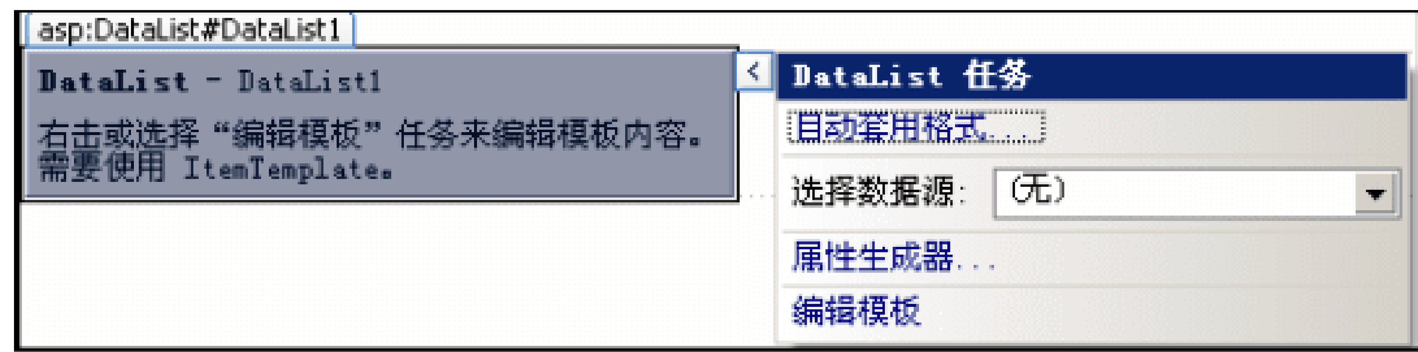


图 3.19　DataList 智能任务面板



图 3.20　DataList 编辑模板视图

当 DataList 处于编辑模板状态时，从外观上看起来就好像一个控件容器，可以从工具箱拖放控件放到这里。这些控件及其布局就是 DataList 的项模板，DataList 数据源中的每一条数据都将使用一组这些控件来显示。在 DataList 智能任务页面有一个下拉列表，里面包含了 DataList 的所有模板，可以从其中选择一种模板进行编辑。图 3.21 所示的 DataList 控件项目模板中放置了 4 个 Label 控件。

在 DataList 控件的智能任务面板中选择“结束模板编辑”选项，以返回默认的 DataList 设计界面，这时的界面就是页面运行时 DataList 的界面，只是其中的数据都是虚拟的。可以看到，DataList 现在变成了具有多行的一个 Table，每一行中都有 4 个 Label，这 4 个 Label 就是项模板中所定义的内容，如图 3.22 所示。

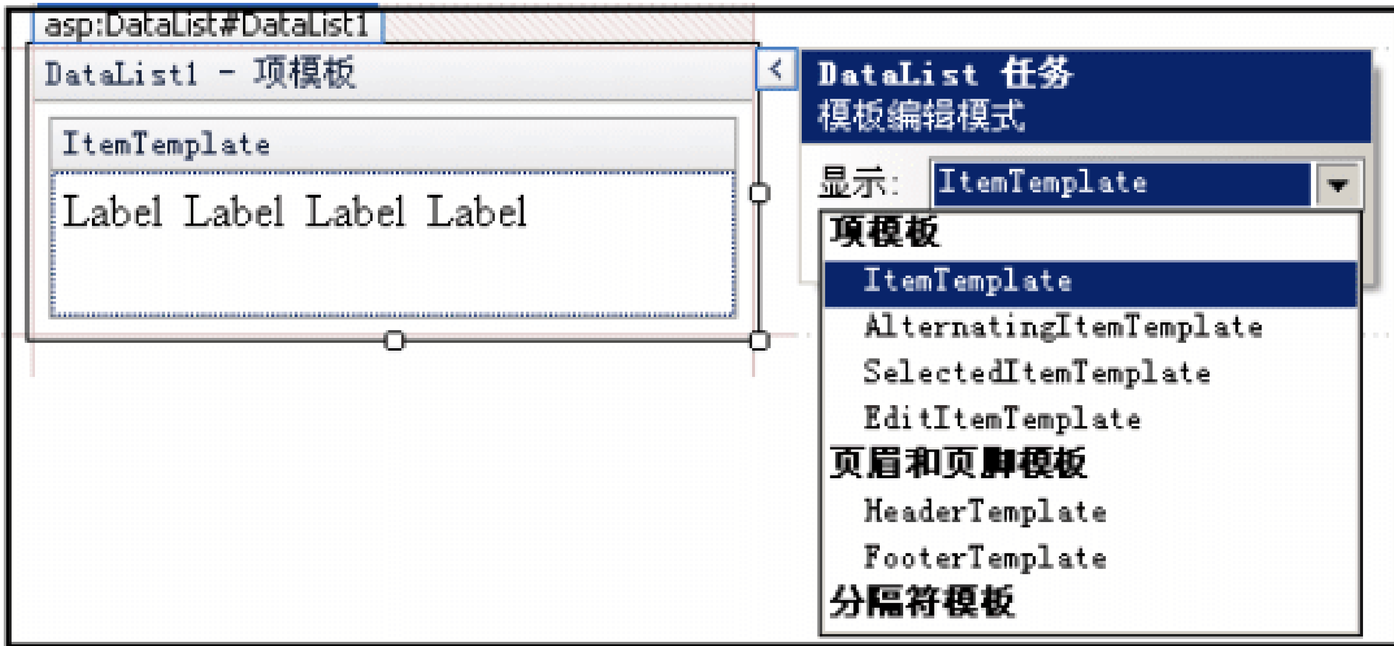


图 3.21　包含了控件的 DataList 项模板



图 3.22　项模板为 4 个 Label 的 DataList 外观

此时切换到页面的代码视图，DataList 控件代码变成如下内容。

```
<asp:DataList ID="DataList1" runat="server">
   <ItemTemplate>
       <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
        
       <asp:Label ID="Label2" runat="server" Text="Label"></asp:Label>
        
       <asp:Label ID="Label3" runat="server" Text="Label"></asp:Label>
        
       <asp:Label ID="Label4" runat="server" Text="Label"></asp:Label>
   </ItemTemplate>
</asp:DataList>
```

（4）到目前为止，已经基本定义好了用来显示数据的控件，下一步需要将控件和要显示的数据联系起来，即将数据绑定到 DataList 模板中的控件。本例中，需要把商品信息各个字段分别绑定到各个 Label 上。操作方法是切换到 DataList 的编辑项模板视图，选中项模板中的一个控件，如第一个 Label，然后打开 Label 的智能任务面板，选择“编辑 DataBindings”命令，即弹出 Label 的数据绑定对话框，如图 3.23 所示。

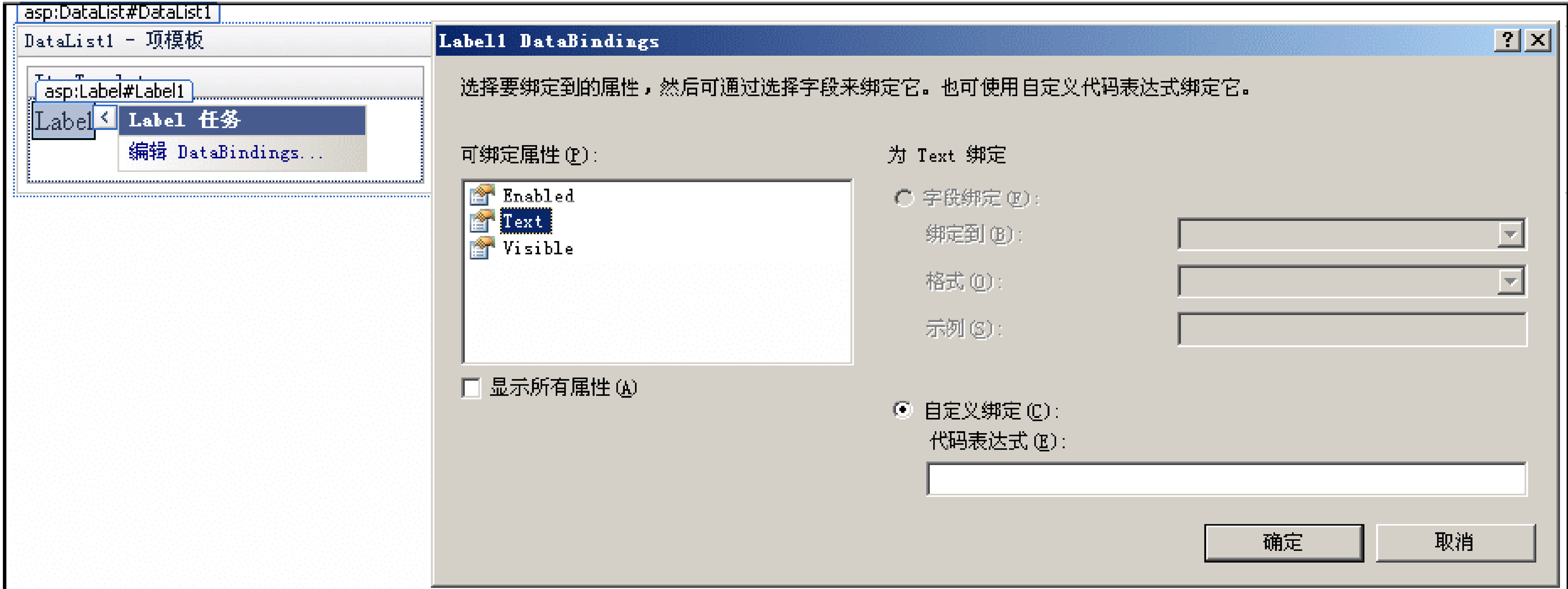


图 3.23　Label 数据绑定对话框

在图 3.21 所示对话框中，从左侧下拉列表框中选择 Text 属性，然后选中右侧的“自定义绑定”RadioButton，在“代码表达式”中填写“Eval("ProductName")”（不包括最外层的中文引号）。其中 Eval 是数据绑定控件的一个方法，作用是求方法参数所指定字段的值，ProductName 为要绑定的字段名。这个操作的作用是用 Label 的 Text 属性设置为数据源中 ProductName 字段的值，即把 ProductName 字段的值显示在 Label 上。

按照同样的操作步骤，为后面 3 个 Label 的 Text 属性设定数据绑定表达式，分别绑定到 Price、Stock 和 Description 字段。所有字段绑定结束后，DataList 代码成为以下内容。

```
<asp:DataList ID="DataList1" runat="server">
   <ItemTemplate>
       <asp:Label ID="Label1" runat="server" Text='<%# Eval("ProductName")
       %>'></asp:Label>
        
       <asp:Label ID="Label2" runat="server" Text='<%# Eval("Price")%>'>
       </asp:Label>  
       <asp:Label ID="Label3" runat="server" Text='<%# Eval("Stock")%>'>
       </asp:Label>  
       <asp:Label ID="Label4" runat="server" Text='<%# Eval("Description")%
       >'></asp:Label>
```

```
   </ItemTemplate>
</asp:DataList>
```

（5）在 PageLoad 事件中，编写代码从数据库读取数据并且绑定到 DataList 控件。

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        SqlHelper db = new SqlHelper();
        DbCommand command = db.GetSqlStringCommond("select * from
        products");
        DataTable table = db.ExecuteDataTable(command);
        DataList1.DataSource = table;
        DataList1.DataBind();
    }
}
```

（6）运行 SimpleDataList.aspx 页面，运行界面如图 3.24 所示。

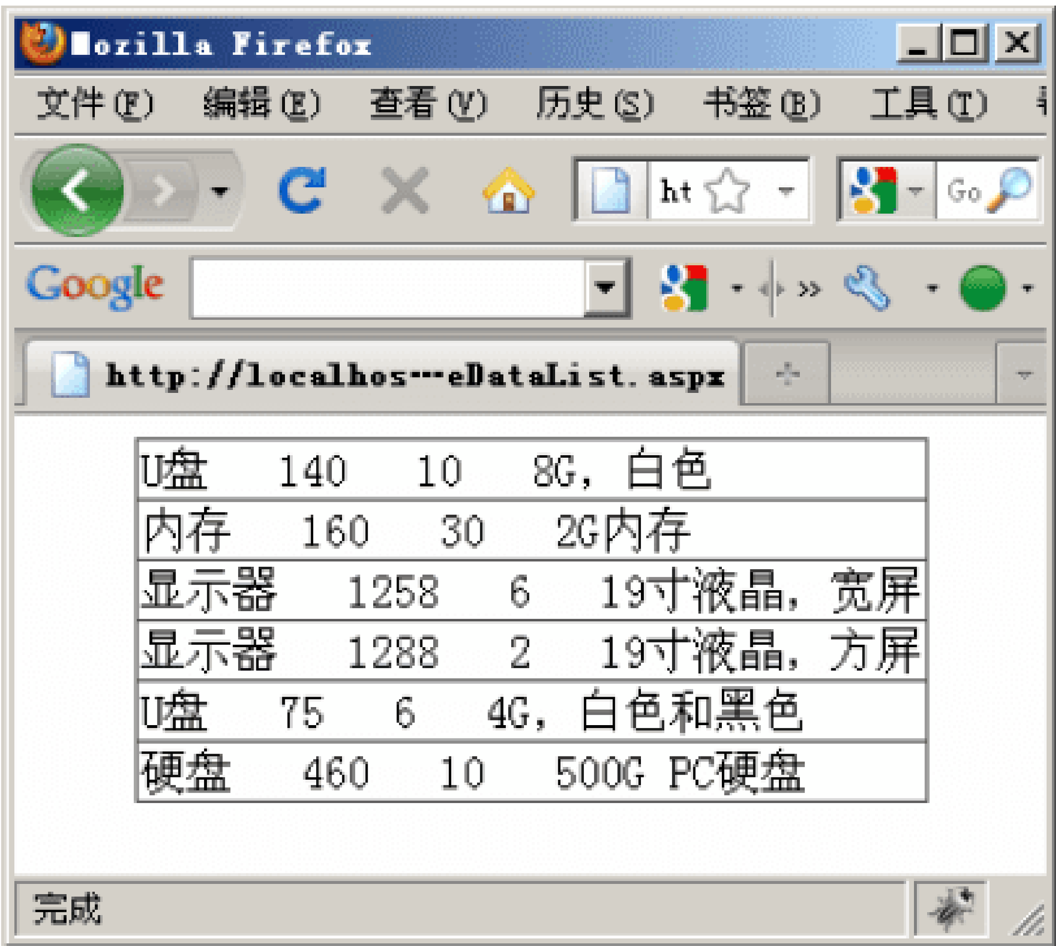


图 3.24　DataList 以表格形式显示数据

在例 3-11 中，DataList 显示出来的数据格式比较凌乱，各行的同一字段没有垂直对齐。出现这种情况的原因可以从页面生成的 HTML 源码中找到。SimpleDataList.aspx 生成的 HTML 源码关键代码如下：

```
<table id="DataList1" cellspacing="0" align="Center" rules="all" border
="1" style="border-collapse:collapse;">
    <tr>
        <td>
            <span id="DataList1 ctl00 Label1">U 盘</span>
             <span id="DataList1 ctl00 Label2">140</span>   
             <span id="DataList1 ctl00 Label3">10</span>   
             <span id="DataList1_ctl00_Label4">8G，白色</span>
          </td>
    </tr><tr>
        <td>
            <span id="DataList1_ctl01_Label1">内存</span>
             <span id="DataList1_ctl01_Label2">160</span>   
             <span id="DataList1 ctl01 Label3">30</span>   
             <span id="DataList1_ctl01_Label4">2G 内存</span>
          </td>
    </tr>
<table>
```

从以上代码可以看出，DataList 控件生成了一个 table，DataList 每一项对应于一个 tr，

但是项模板里每个控件并不分别对应一个 td，而是所有控件在同一个 td 中，从而就形成了比较混乱的布局。如果能够把项模板中每一个 Label 单独放到一个 td 中，那么就可以实现整齐的数据表格了。下面通过一个例子演示这种思路。

【例 3-12】 以规范的表格显示数据。

本例演示使用 DataList 控件以整齐的带表头的表格显示商品信息列表。

（1）创建一个 ASP.NET 项目，按照例 3-11 的步骤添加数据库。

（2）添加一个新页面 SimpleDataList2.aspx，在页面上放置一个 DataList 控件。

（3）在代码视图中编辑 DataList 控件的头模板，使其生成商品信息表头。代码如下，注意其中第一个 td 元素没有开始标记只有结束标记，而最后一个 td 元素没有结束标记只有开始标记。这是因为 DataList 会自动生成一对<tr><td></td></tr>标记，模板中的内容放在这一对标记中，通过把模板内容设置为以下代码，模板代码与自动生成的代码组合在一起，形成了完整的 5 个 td。

```
<asp:DataList ID="DataList1" runat="server" RepeatLayout="Flow" >
<HeaderTemplate>
商品编号</td>
<td>商品名称</td>    <td>商品价格</td>    <td>商品库存</td>
<td>商品描述
</HeaderTemplate>
</asp:DataList>
```

（4）在代码视图中编辑 DataList 控件的项模板，项中每一个 Label 都在一个 td 中。同样的道理，第一个 td 元素没有开始标记只有结束标记，而最后一个 td 元素没有结束标记只有开始标记。

```
<ItemTemplate>
 <asp:Label ID="Label0" runat="server" Text='<%# Eval("ProductID")%>'>
</asp:Label></td>
   <td> <asp:Label ID="Label1" runat="server" Text='<%# Eval("Product-
   Name")%>'> </asp:Label> </td>
   <td><asp:Label ID="Label2" runat="server" Text='<%# Eval("Price")%>'
   ></asp:Label></td>
   <td><asp:Label ID="Label3" runat="server" Text='<%# Eval("Stock")%>
   '></asp:Label></td>
   <td><asp:Label ID="Label4" runat="server" Text='<%# Eval("Description
   ")%>'></asp:Label>
</ItemTemplate>
```

（5）在 Page_Load 事件中编写代码以绑定数据，代码同前。

（6）运行 SimpleDataList2.aspx 页面，运行效果如图 3.25 所示。

图 3.25　DataList 以表格形式显示数据

3.3.2　自定义布局

DataList 控件可以自定义数据显示布局，从而设计出丰富美观的页面。在购物网站经常看到的商品列表页面，既有商品的各种图片，也有商品的相关文字信息，用 DataList 控件能够很方便地开发出这种图文并茂的页面。

【例 3-13】 显示商品列表。

本例用 DataList 控件以图文并茂的形式显示商品列表。

（1）创建一个 ASP.NET 应用程序，把前面所创建的包含 Products 表的数据库复制到项目中 App_Data 目录下。

（2）修改 Products 表结构，添加一列 ImageUrl，类型为 nvarchar(50)，表示商品图片路径。在项目中添加一个 ProductImages 文件夹，用于存放商品图片。

（3）在 Products 里添加几种商品，把商品图片复制到 ProductImages 文件夹，并且设置 Products 表中 ImageUrl 这一列的值。

（4）在项目中添加一个页面 ProductList.aspx，在页面上放置一个 DataList 控件。

（5）本例同时显示商品图片和文字信息，图片显示在左侧，文字在右侧。可以用一个 table 来实现这种布局，按照这种思路设置 DataList 控件的项模板，代码如下：

```
<asp:DataList ID="DataList1" runat="server" HorizontalAlign="Center">
<ItemTemplate>
<table> <tr>
<td> <img src="ProductImages/<%#Eval("ImageUrl")%>"
alt="<%#Eval("ProductName") %>" style="width:100px;" />
</td>
<td><span style="font-weight:bold;"><%#Eval("ProductName")%></span><br />
价格：<%#Eval("Price","{0:C}") %><br />
说明：<%#Eval("Description") %> </td>
</tr>   </table>
</ItemTemplate>
<ItemStyle BackColor="White" />
<AlternatingItemStyle BackColor="#ffffcc" />
</asp:DataList>
```

（6）在 Page_Load 事件中，编写代码将数据绑定到 DataList。

（7）运行 ProductList.aspx 页面，运行效果如图 3.26 所示。

（8）由于商品信息太少，图 3.26 所示的页面显得很细长，不够美观，而且浪费了页面很多空间。如果能够在将 2 种或者 3 种商品信息合并在一行显示，那么效果就会更好。DataList 控件支持这个功能，通过设置其 RepeatColumns 属性可以指定每一行中显示的项数目。此处将 RepeatColumns 设置为 2，代码如下：

```
<asp:DataList ID="DataList1" runat="server" HorizontalAlign="Center"
RepeatColumns="2">
```

（9）再次运行 ProductList.aspx 页面，运行效果如图 3.27 所示。

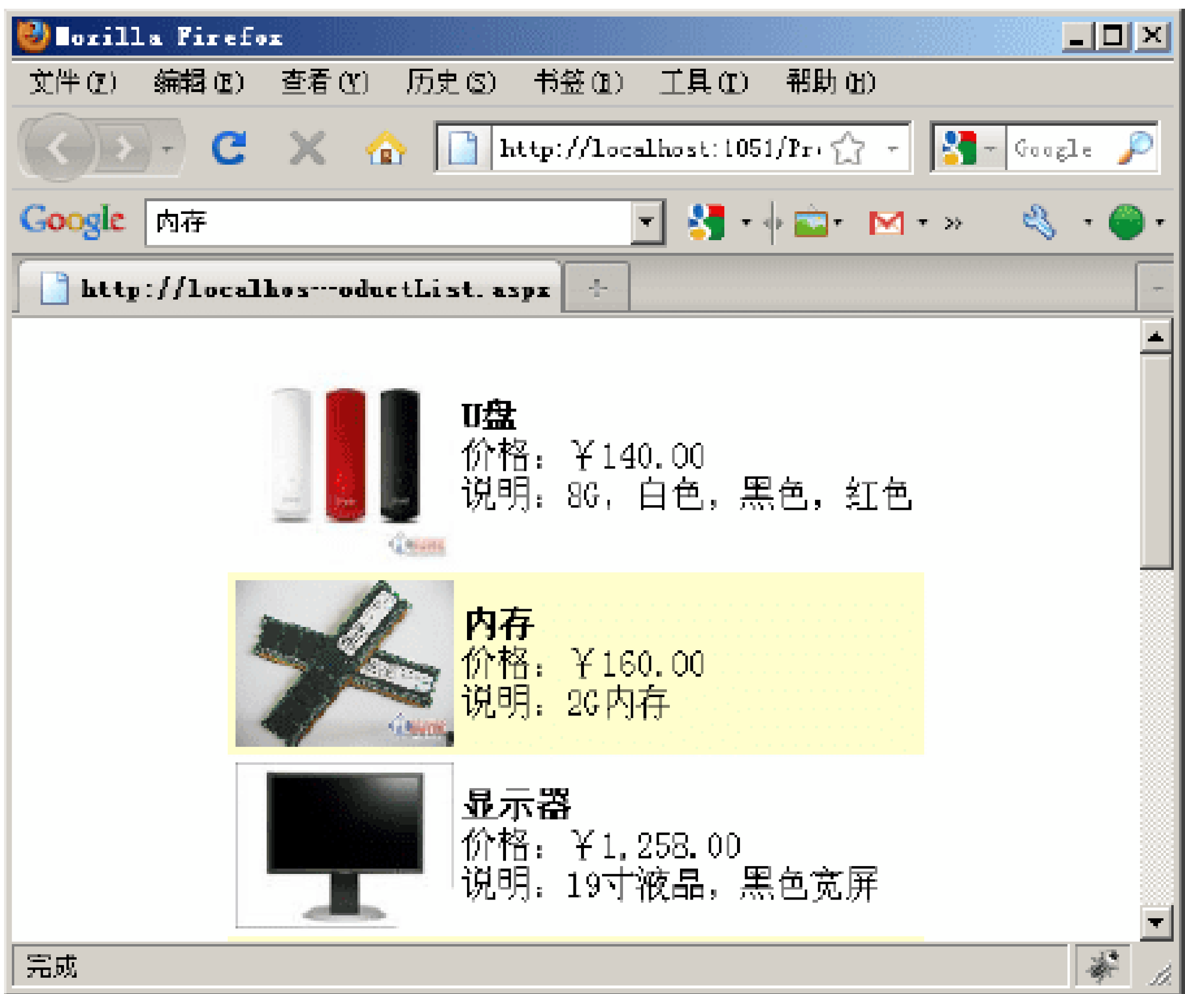


图 3.26　商品列表页面



图 3.27　每行 2 种商品的商品列表

3.3.3　DataList 编辑数据

用 GridView 控件实现数据编辑和删除时，只需要添加一个 CommandField，再配合后台代码即可。而 DataList 控件不支持 CommandField，所以 DataList 实现数据删除和编辑要比 GridView 稍微复杂一些。

【例 3-14】 DataList 编辑和删除数据。

本例演示在用 DataList 显示、编辑和删除商品列表。

（1）打开例 3-13 所创建的项目，或者新建一个 ASP.NET 项目，把例 3-13 的数据库和图片文件复制到新项目中。

（2）在项目中添加一个新页面 DataListEdit.aspx。

（3）为 DataList 创建项模板以显示商品列表。本例使用的项模板与例 3-13 不同，例 3-13 每种商品用一个单独的 table 显示，而本例所有商品使用同一个 table 显示，每种商品是 table 中的一个 tr。为了实现这种效果，需要定制 DataList 控件的 HeaderTemplate，在 ItemTemplate 中也要注意项模板中 td 标记与 DataList 自动生成的 td 元素的结合。另外，DataList 中还包含两个 LinkButton 列，分别用于编辑和删除数据。

```
<asp:DataList ID="DataList1" runat="server" HorizontalAlign="Center"
   oneditcommand="DataList1_EditCommand" DataKeyField="ProductID"
   oncancelcommand="DataList1_CancelCommand"
   ondeletecommand="DataList1_DeleteCommand"
   onupdatecommand="DataList1_UpdateCommand">
   <HeaderTemplate>
  </td><td></td><td></td><td>
   </HeaderTemplate>
<ItemTemplate>
<img src="ProductImages/<%#Eval("ImageUrl")%>" alt="<%#Eval("ProductName")
%>" style="width:100px;" />  </td>
<td><span style="font-weight:bold;"> <%#Eval("ProductName")%> </span><br
/>
价格：<%#Eval("Price","{0:C}") %><br />说明：<%#Eval("Description") %></td>
<td><asp:LinkButton runat="server" id="editButton" Text="编辑" Command-
Name="edit" /></td>
<td>
 <asp:LinkButton runat="server" id="deleteButton" Text="删除" CommandName
="delete"
 OnClientClick="return confirm('确实要删除吗？');" />
</ItemTemplate>
</asp:DataList>
```

在上述代码中，在“删除”按钮中用了一个小技巧。“删除”按钮在客户端会调用 JavaScript 语句弹出一个对话框，请用户确认是否删除。如果用户选择取消删除，则不会产生服务器回发，也不会删除数据。

（4）为了实现 DataList 编辑数据的功能，需要定制 EditItemTemplate 模板。在模板中，用 TextBox 控件显示商品名称、单价、库存、描述信息，并允许用户编辑，用文件上传控件 FileUpload 接受用户上传的产品图片文件。在编辑项模板 EditItemTemplate 中，“编辑”按钮变成了两个按钮：一个“取消”和一个“更新”。

```
<EditItemTemplate>
    上传图片文件：<br />
    <asp:FileUpload ID="imageFile" runat="server" /><br />
    (如果不上传则保持原有图片不变)
    </td>
<td>
名称：
<asp:TextBox runat="server" ID="productName" Text='<%#Eval("ProductName")
%>' /> <br />
价格：<asp:TextBox runat="server" ID="productPrice" Text='<%#Eval("Price")
%>' /> <br />
说明：<asp:TextBox runat="server" ID="description" Text='<%#Eval(
"Description")%>' /> <br />
库存：<asp:TextBox runat="server" ID="stock" Text='<%#Eval("Stock")%>' />
</td> <td>
<asp:LinkButton runat="server" id="cancelButton" Text="取消" CommandName
="cancel"  />
<asp:LinkButton runat="server" id="updateButton" Text="保存" CommandName
="update"  />
</td>
<td> <asp:LinkButton runat="server" id="deleteButton" Text='删除'
CommandName="delete"
 OnClientClick="return confirm('确实要删除吗？');" />
</EditItemTemplate>
```

（5）在 Page_Load 事件中编写代码绑定数据。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindList();
    }
}
private void bindList()
{
    //读取商品数据并绑定到 DataList
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond("select * from products");
    DataTable table = db.ExecuteDataTable(command);
    DataList1.DataSource = table;
    DataList1.DataBind();
}
```

（6）在 DataList 的 EditCommand 事件中，将事件发生的项设置为编辑状态。

```
protected void DataList1 EditCommand(object source, DataListCommand
EventArgs e)
{
    DataList1.EditItemIndex = e.Item.ItemIndex;             //设置当前编辑项
    bindList();
}
```

（7）在 DataList 的 CancelCommand 事件中，取消所有项的编辑状态。

```
protected void DataList1 CancelCommand(object source, DataListCommand-
EventArgs e)
{
    DataList1.EditItemIndex = -1;                           //取消编辑状态
    bindList();
}
```

（8）在 DataList 的 DeleteCommand 事件中，删除当前项。

```
protected void DataList1_DeleteCommand(object source, DataListCommand-
EventArgs e)
{
    string key = DataList1.DataKeys[e.Item.ItemIndex].ToString();
                                                            //得到主键
    //如果要删除的项当前处于编辑状态，则先取消编辑状态
    if (DataList1.EditItemIndex == e.Item.ItemIndex)
        DataList1.EditItemIndex = -1;
    string sql = "delete from products where productid=@id";
                                                            //构建 delete 语句
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond(sql);    //创建一个命令
    db.AddInParameter(command, "@id", DbType.String, key);
         //向命令添加参数
    db.ExecuteNonQuery(command);                            //执行 delete 命令
    bindList();                                             //重新绑定数据
}
```

（9）在 DataList 的 UpdateCommand 事件中，将用户输入的数据保存到数据库。要注意其中对于商品图片文件的处理。

```
protected void DataList1_UpdateCommand(object source, DataListCommand-
EventArgs e)
{
    //获取用户输入的数据
    string name = (e.Item.FindControl("productName") as TextBox).Text;
    double price=double.Parse((e.Item.FindControl("productPrice") as
    TextBox).Text);
    double stock = double.Parse((e.Item.FindControl("stock") as TextBox)
    .Text);
    string description = (e.Item.FindControl("description") as TextBox).
    Text;
    string key = DataList1.DataKeys[e.Item.ItemIndex].ToString();
    //获得上传文件
    FileUpload file = e.Item.FindControl("imageFile") as FileUpload;
    string image = "";
    //如果用户传了文件则保存到 ProductImage 目录下
    if (file != null && file.HasFile)
    {
        file.SaveAs(Server.MapPath("~/ProductImages/" + file.FileName));
        image = file.FileName;
    }
    //构建 update 语句
    string sql = "update Products set ProductName=@name, Price=@price,"
        + " Stock=@stock, Description=@description "
        + (file.HasFile?", ImageUrl=@img ":" ")
        + " where ProductID=@id";
    //创建一个命令以包含刚才创建的 update 语句
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond(sql);
    //向 update 命令添加各个参数
    db.AddInParameter(command, "@name", DbType.String, name);
    db.AddInParameter(command, "@price", DbType.Double, price);
    db.AddInParameter(command, "@stock", DbType.Double, stock);
    db.AddInParameter(command, "@description", DbType.String, description);
    db.AddInParameter(command, "@id", DbType.String, key);
```

```
    if(file.HasFile)
        db.AddInParameter(command, "@img", DbType.String,image);
    //执行 update 命令，取消当前项的编辑状态，重新绑定数据
    db.ExecuteNonQuery(command);
    DataList1.EditItemIndex = -1;
    bindList();
}
```

（10）运行 DataListEdit.aspx 页面，页面运行界面如图 3.28 所示。

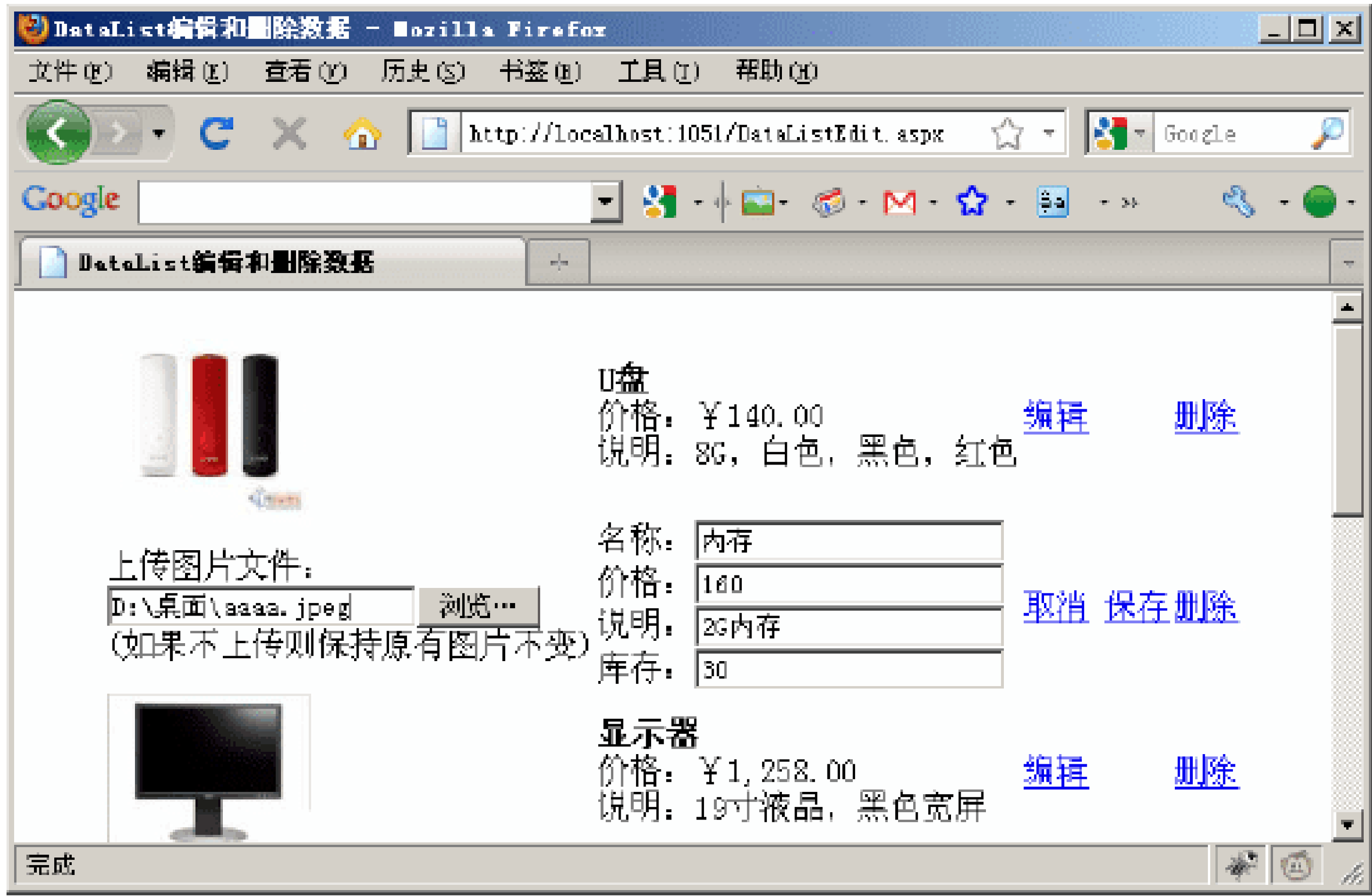


图 3.28　用 DataList 显示和编辑商品列表

3.4　其他数据绑定控件

前面两节分别介绍了 GridView 和 DataList 控件，这也是 ASP.NET 中最常用的两种数据控件。其他数据绑定控件还包括 Repeater、FormView、DetailsView 等，有了前面的基础，这几种控件的使用也很容易掌握。

3.4.1　Repeater 控件

Repeater 控件与 DataList 类似，也是以数据列表的形式显示数据，也需要自定义项模板。二者的不同之处在于，在最终生成的 HTML 页面中，DataList 控件会自动生成少量布局代码，如 table、tr、br，而 Repeater 控件本身不会生成任何 HTML 代码，所以 Repeater 通常用于需要严格控制生成的 HTML 的情况下。下面通过一个例子来演示 Repeater 控件的使用，这个例子的功能也是显示商品列表，通过将这个例子与前面 DataList 显示商品列表的例子比较，可以更明确地体会二者的异同。

【例 3-15】 Repeater 控件显示商品列表。

（1）打开例 3-13 所创建的项目，或者新建一个 ASP.NET 项目，把例 3-13 的数据库和图片文件复制到新项目中。

（2）添加一个新页面 RepeaterSample.aspx，在页面上放置一个 Repeater 控件。

（3）本例要用一个 table 来显示所有商品信息，每个商品作为一个 tr。在 Repeater 控件的 HeaderTemplate 中，开始 table 标记。

```
<HeaderTemplate>        <table align="center">        </HeaderTemplate>
```

（4）在 Repeater 控件的 FooterTemplate 中，结束 table 标记。

```
<FooterTemplate>        </table>        </FooterTemplate>
```

（5）在 Repeater 控件的项模板 ItemTemplate 中，生成一个 tr 元素显示商品信息，其中包括两个 td，商品图片为第一个 td，其他文字信息为第二个 td。要注意其中表示是否有货的 CheckBox 控件的数据绑定表达式。

```
<ItemTemplate>
<tr>    <td>
<img src="ProductImages/<%#Eval("ImageUrl") %>" alt="<%#Eval("Product-
Name") %>" style="width:100px;" /></td>
<td><span style="font-weight:bold;"><%#Eval("ProductName")%></span><br />
价格：<%#Eval("Price","{0:C}") %><br />说明：<%#Eval("Description") %><br />
是否有货：<asp:CheckBox runat="server" Enabled="false"
Checked='<%#double.Parse(Eval("Stock").ToString())>0 %>'  />
</td>    </tr>
</ItemTemplate>
```

（6）在 Page_Load 事件中，编写代码绑定数据。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        //查询所有商品数据并绑定到 Repeater 控件
        SqlHelper db = new SqlHelper();
        DbCommand command = db.GetSqlStringCommond("select * from products");
        DataTable table = db.ExecuteDataTable(command);
        Repeater1.DataSource = table;
        Repeater1.DataBind();
    }
}
```

（7）运行 RepeaterSample.aspx 页面，运行效果如图 3.29 所示。

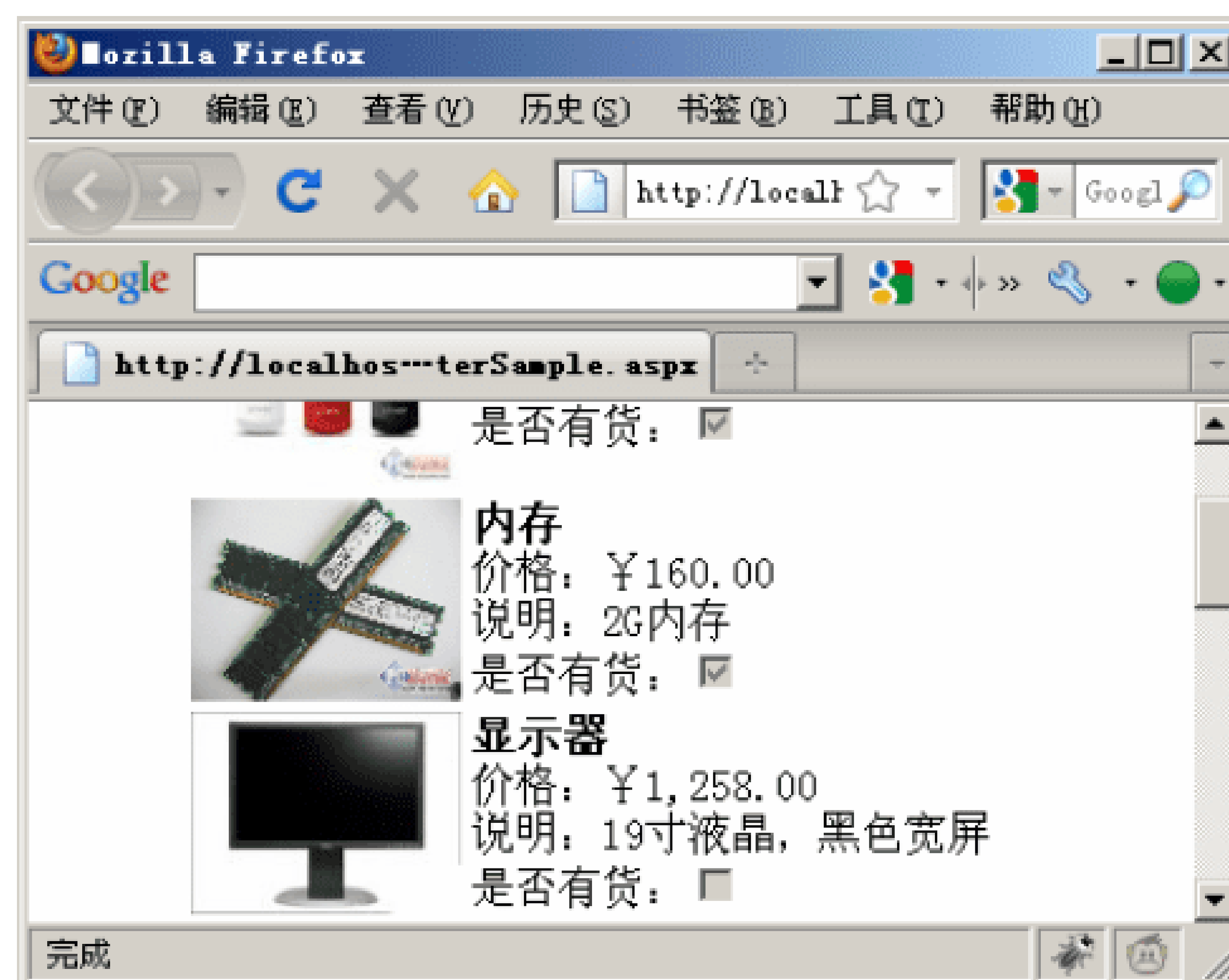


图 3.29　Repeater 控件显示商品列表

3.4.2　DetailsView 控件

正如其名称所指示，DetailsView 用来显示数据细节。DetailsView 控件用表格的形式

显示数据，每次只能显示一条数据，表中一行对应于数据源的一个字段。DetailsView 支持分页，如果数据源中有多条数据，那么在 DetailsView 中就需要使用多页显示。DetailsView 通常用来显示数据详情或者编辑、插入数据。DetailsView 有 3 个显示模式，分别是只读模式、编辑模式和插入模式，可以编程对这 3 种模式进行切换。

【例 3-16】 DeteailsView 显示和编辑数据。

本例演示使用 DetailsView 显示、编辑、删除、添加商品信息。

（1）打开例 3-13 所创建的项目，或者新建一个 ASP.NET 项目，把例 3-13 的数据库和图片文件复制到新项目中。

（2）添加一个新页面 DetailsViewSample.aspx，在页面上放置一个 DetailsView 控件。

（3）将 DetailsView 控件的 AutoGenerateDeleteButton、AutoGenerateEditButton、AutoGenerateInsertButton 属性都设置为 true，使 DetailsView 自动生成添加、删除和编辑按钮。将 AllowPaging 属性设置为 true 以允许分页。此时 DetailsView 代码如下：

```
<asp:DetailsView ID="detail" runat="server" DataKeyNames="ProductID"
    AllowPaging="true"
    AutoGenerateRows="false" AutoGenerateDeleteButton="true"
    AutoGenerateEditButton="true" AutoGenerateInsertButton="true"
    onpageindexchanging="detail_PageIndexChanging"
    onitemdeleting="detail_ItemDeleting"
    oniteminserting="detail_ItemInserting"
    onitemupdating="detail_ItemUpdating"
    onmodechanging="detail_ModeChanging" >
</asp:DetailsView>
```

（4）为 DetailsView 添加各个字段，设置各字段的项模板和项编辑模板，其中商品编号 ProductID 为只读字段，商品图片使用 FileUpload 控件编辑。完成这些设置后 DetailsView 控件代码如下：

```
<asp:DetailsView ID="detail" runat="server" DataKeyNames="ProductID"
    AllowPaging="true"
    AutoGenerateRows="false" AutoGenerateDeleteButton="true"
    AutoGenerateEditButton="true" AutoGenerateInsertButton="true"
    onpageindexchanging="detail_PageIndexChanging"
    onitemdeleting="detail_ItemDeleting"
    oniteminserting="detail_ItemInserting"
    onitemupdating="detail_ItemUpdating"
    onmodechanging="detail_ModeChanging" >
<%--Fields 标记中为 DetailsView 的字段列表--%>
<Fields>
    <%--asp:Bound 表示数据绑定列，DataField 为要绑定的字段名，ReadOnly 表示只读
--%>
   <asp:BoundField DataField="ProductID" ReadOnly="true" HeaderText="商品
   编号" />
   <asp:BoundField DataField="ProductName"  HeaderText="商品名称" />
   <asp:BoundField DataField="Price" HeaderText="价格" />
   <asp:BoundField DataField="Stock" HeaderText="库存" />
   <%--商品图片是一个模板字段，使用 TemplateField 标记--%>
   <asp:TemplateField HeaderText="商品图片">
      <%--ItemTemplate 为正常显示时使用的模板--%>
      <ItemTemplate>
         <img src="ProductImages/<%#Eval("ImageUrl") %>" alt="" style=
         "width:200px;" />
```

```
        </ItemTemplate>
        <%--EditItemTemplate 为编辑状态时使用的模板--%>
        <EditItemTemplate>
            <asp:FileUpload ID="imageFile" runat="server"  />
        </EditItemTemplate>
    </asp:TemplateField>
</Fields>
</asp:DetailsView>
```

（5）在 Page_Load 事件中编写代码绑定数据。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindList();
    }
}
private void bindList()
{
    //读取所有商品数据并绑定到 DetailsView 控件
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond("select * from products");
    DataTable table = db.ExecuteDataTable(command);
    detail.DataSource = table;
    detail.DataBind();
}
```

（6）在 DetailsView 控件的 PageIndexChanging 事件中，编写代码实现翻页。

```
protected void detail PageIndexChanging(object sender, DetailsViewPage-
EventArgs e)
{
    detail.PageIndex = e.NewPageIndex;                    //切换到新页
    bindList();                                           //重新绑定数据
}
```

（7）在 DetailsView 控件的 ModeChanging 事件中实现 DetailsView 模式切换。

```
protected void detail_ModeChanging(object sender, DetailsViewModeEventArgs
e)
{
    detail.ChangeMode(e.NewMode);
    bindList();
}
```

（8）在 DetailsView 控件的 ItemDeleting 事件中，实现数据删除。

```
protected void detail_ItemDeleting(object sender, DetailsViewDelete-
EventArgs e)
{
    string key = detail.DataKey[0].ToString();          //得到主键
    string sql = "delete from products where productid=@id";
                                                        //构建删除 SQL 语句
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond(sql);
    db.AddInParameter(command, "@id", DbType.String, key);
    db.ExecuteNonQuery(command);
    detail.ChangeMode(DetailsViewMode.ReadOnly);        //切换到只读模式
    bindList();
}
```

（9）在 DetailsView 控件的 ItemInserting 事件中，实现向数据库中添加数据。

```
protected void detail_ItemInserting(object sender, DetailsViewInsert-
EventArgs e)
{
    //得到用户输入的数据
    string id = Convert.ToString(e.Values["ProductID"]);
    string name = Convert.ToString(e.Values["ProductName"]);
    double price = Convert.ToDouble(e.Values["Price"]);
    double stock = Convert.ToDouble(e.Values["Stock"]);
    string description = Convert.ToString(e.Values["Description"]);
    string image = "";
    FileUpload file = detail.FindControl("imageFile") as FileUpload;
    //如果用户上传了文件，则将文件保存到服务器
    if (file.HasFile)
    {
        image = file.FileName;
        file.SaveAs(Server.MapPath("~/ProductImages/" + file.FileName));
    }
    //构建 SQL 语句
    string sql = "insert into Products "
        + " (ProductID, ProductName, Price, Stock, Description, ImageUrl)"
        + " values(@id, @name, @price, @stock, @description, @img)";
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond(sql);
    //向 SqlCommand 对象添加各个参数
    db.AddInParameter(command, "@name", DbType.String, name);
    db.AddInParameter(command, "@price", DbType.Double, price);
    db.AddInParameter(command, "@stock", DbType.Double, stock);
    db.AddInParameter(command, "@description", DbType.String, description);
    db.AddInParameter(command, "@id", DbType.String, id);
    db.AddInParameter(command, "@img", DbType.String, image);
    db.ExecuteNonQuery(command);                    //执行 insert 命令
    detail.ChangeMode(DetailsViewMode.ReadOnly); //detailsview 设置为只读模式
    bindList();                                     //重新绑定数据
}
```

（10）在 DetailsView 的 ItemUpdating 事件中，把用户对商品信息的修改更新到数据库。

```
protected void detail ItemUpdating(object sender, DetailsViewUpdate-
EventArgs e)
{
    //读取用户输入的数据
    string id = detail.DataKey[0].ToString();
    string name = Convert.ToString(e.NewValues["ProductName"]);
    double price = Convert.ToDouble(e.NewValues["Price"]);
    double stock = Convert.ToDouble(e.NewValues["Stock"]);
    string description = Convert.ToString(e.NewValues["Description"]);
    string image = null;
    FileUpload file = detail.FindControl("imageFile") as FileUpload;
    //如果用户选择了上传文件，则将文件保存到服务器
    if (file.HasFile)
    {
        image = file.FileName;
        file.SaveAs(Server.MapPath("~/ProductImages/" + file.FileName));
    }
    //构建 update 语句
    string sql = "update Products set ProductName=@name, Price=@price,"
        + " Stock=@stock, Description=@description "
```

```
        + (file.HasFile ? ", ImageUrl=@img " : " ")
        + " where ProductID=@id";
    //创建一个 DbCommand 以封装 update 语句，并添加各个参数
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond(sql);
    db.AddInParameter(command, "@name", DbType.String, name);
    db.AddInParameter(command, "@price", DbType.Double, price);
    db.AddInParameter(command, "@stock", DbType.Double, stock);
    db.AddInParameter(command, "@description", DbType.String, description);
    db.AddInParameter(command, "@id", DbType.String, id);
    if (file.HasFile)
        db.AddInParameter(command, "@img", DbType.String, image);
    //执行 update 命令，重新绑定数据
    db.ExecuteNonQuery(command);
    detail.ChangeMode(DetailsViewMode.ReadOnly);
    bindList();
}
```

（11）运行 DetailsViewSample.aspx 页面，运行效果如图 3.30 所示。

3.4.3　FormView 控件

FormView 控件与 DetailsView 控件比较类似，也是用于显示数据源中的单个记录。FormView 控件和 DetailsView 控件之间的差别在于 DetailsView 控件使用表格布局，记录的每个字段都各自显示为一行。而 FormView 控件不指定用于显示记录的预定义布局，而将创建一个模板，其中包含用于显示记录中的各个字段的控件。

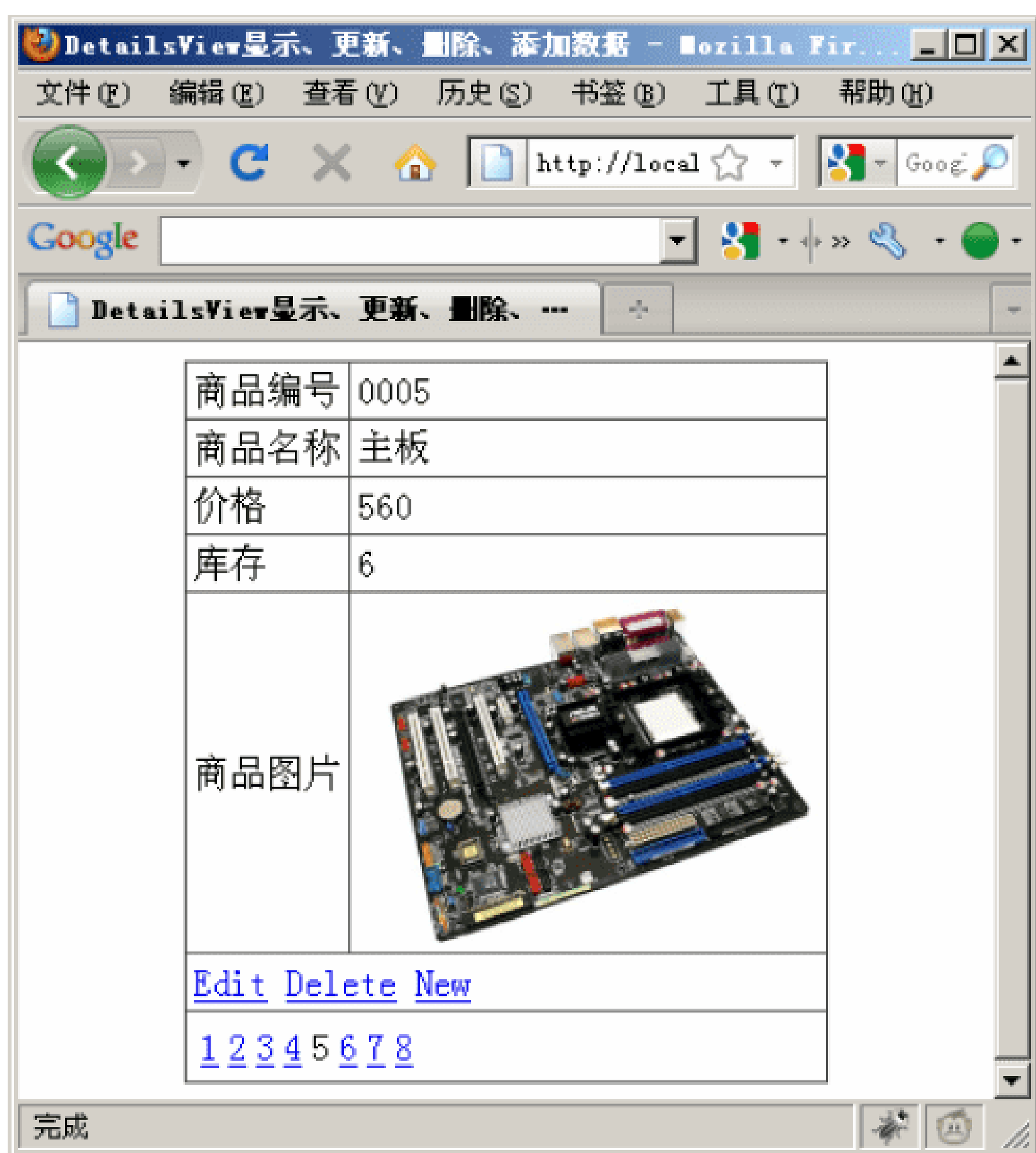


图 3.30　DetailsView 示例

【例 3-17】 FormView 显示、编辑、删除数据。

本例演示使用 FormView 控件显示、编辑和删除商品信息。

（1）打开例 3-13 所创建的项目，或者新建一个 ASP.NET 项目，把例 3-13 的数据库和图片文件复制到新项目中。

（2）添加一个新页面 FormViewSample.aspx，在页面上放置一个 FormView 控件。

（3）配置 FormView 的项模板，使用表格形式显示商品图片和信息。

```
<ItemTemplate>
   <table>
   <tr><td>
   <img src="ProductImages/<%#Eval("ImageUrl") %>" alt="<%#Eval("
ProductName") %>" style="width:200px;" /></td>
   <td> <span style="font-weight:bold;"><%#Eval ("ProductName")%></
   span><br />
    价格：<%#Eval("Price","{0:C}") %><br />说明：<%#Eval("Description") %><br
    />库存：<%#Eval("Stock") %></td>
    <td> <asp:LinkButton runat="server" id="editButton" Text="编辑"
    CommandName="edit"  />
```

```
    </td> <td>
    <asp:LinkButton runat="server" id="deleteButton" Text="删除"  Command-
    Name="delete"
     OnClientClick="return confirm('确实要删除吗？');"  />
    </td> </tr>
</table>
</ItemTemplate>
```

（4）配置 FormView 的编辑项模板，用 TextBox 控件编辑商品文字信息，用 FileUpload 控件上传图片文件。

```
<EditItemTemplate>
    <table>  <tr>  <td>
    商品图片<br />
    <img  src="ProductImages/<%#Eval("ImageUrl") %>" alt="<%#Eval(
    "ProductName") %>" style="width:200px;" /><br />
    上传新的商品图片：<br />
        <asp:FileUpload ID="imageFile" runat="server" /><br />
        (如果不上传则保持原有图片不变)    </td>    <td>
    名称：<asp:TextBox runat="server" ID="productName" Text='<%#Eval
     ("ProductName")%>' /> <br />
    价格：<asp:TextBox runat="server" ID="productPrice" Text='<%#Eval
    ("Price")%>' /> <br />
    说明：<asp:TextBox runat="server" ID="description" Text='<%#Eval
    ("Description")%>' /> <br />
    库存：<asp:TextBox runat="server" ID="stock" Text='<%#Eval("Stock")%>'
    /> </td> <td>
    <asp:LinkButton runat="server" id="cancelButton" Text="取消"
    CommandName="cancel"  />
    <asp:LinkButton runat="server" id="updateButton" Text="保存" Command-
    Name="update"  />
    </td>  <td>
    <asp:LinkButton runat="server" id="deleteButton" Text="删除"
    CommandName="delete"
     OnClientClick="return confirm('确实要删除吗？');" />
    </td> </tr>
    </table>
</EditItemTemplate>
```

（5）在 Page_Load 事件中绑定数据列表。

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindList();
    }
}
private void bindList()
{
    //读取商品数据并绑定到 FormView 控件
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond("select * from products");
    DataTable table = db.ExecuteDataTable(command);
    formView1.DataSource = table;
    formView1.DataBind();
}
```

（6）在 FormView 控件的 PageIndexChanging 事件中，编写代码实现翻页。

```
protected void formView1_PageIndexChanging(object sender, FormViewPage-
EventArgs e)
{
    formView1.PageIndex = e.NewPageIndex;
    bindList();
}
```

（7）在 FormView 控件的 ModeChanging 事件中实现 FormView 模式切换。

```
protected void formView1_ModeChanging(object sender, FormViewModeEventArgs
e)
{
    formView1.ChangeMode(e.NewMode);
    bindList();
}
```

（8）在 FormView 控件的 ItemDeleting 事件中，实现数据删除。

```
protected void formView1 ItemDeleting(object sender, FormViewDelete-
EventArgs e)
{
    string key = formView1.DataKey[0].ToString();        //得到主键值
    string sql = "delete from products where productid=@id";
                                                          //构建 delete 语句
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond(sql);
    db.AddInParameter(command, "@id", DbType.String, key);
    db.ExecuteNonQuery(command);                          //执行删除
    formView1.ChangeMode(FormViewMode.ReadOnly);
    bindList();                                           //重新绑定
}
```

（9）在 FormView 控件的 ItemUpdating 事件中，得到用户输入的信息，并保存到数据库。

```
protected void formView1 ItemUpdating(object sender, FormViewUpdate-
EventArgs e)
{
    //获取用户输入的数据
    string id = formView1.DataKey[0].ToString();
    string name = (formView1.FindControl("productName") as TextBox).Text;
    double price = double.Parse((formView1.FindControl("productPrice") as
    TextBox).Text);
    double stock = double.Parse((formView1.FindControl("stock") as
    TextBox).Text);
    string description = (formView1.FindControl("description") as TextBox).
    Text;
    string image = null;
    FileUpload file = formView1.FindControl("imageFile") as FileUpload;
    //如果用户上传了文件，则将其保存到服务器
    if (file.HasFile)
    {
        image = file.FileName;
        file.SaveAs(Server.MapPath("~/ProductImages/" + file.FileName));
    }
    //构建 update 语句
    string sql = "update Products set ProductName=@name, Price=@price,"
        + " Stock=@stock, Description=@description "
        + (file.HasFile ? ", ImageUrl=@img " : " ")
```

```
        + " where ProductID=@id";
    //创建一个命令对象以封装 update 语句，并对该命令对象添加各个参数
    SqlHelper db = new SqlHelper();
    DbCommand command = db.GetSqlStringCommond(sql);
    db.AddInParameter(command, "@name", DbType.String, name);
    db.AddInParameter(command, "@price", DbType.Double, price);
    db.AddInParameter(command, "@stock", DbType.Double, stock);
    db.AddInParameter(command, "@description", DbType.String, description);
    db.AddInParameter(command, "@id", DbType.String, id);
    if (file.HasFile)
        db.AddInParameter(command, "@img", DbType.String, image);
    //执行 update 语句，然后重新绑定数据
    db.ExecuteNonQuery(command);
    formView1.ChangeMode(FormViewMode.ReadOnly);
    bindList();
}
```

（10）运行 FormViewSample.aspx 页面，运行效果如图 3.31 所示。

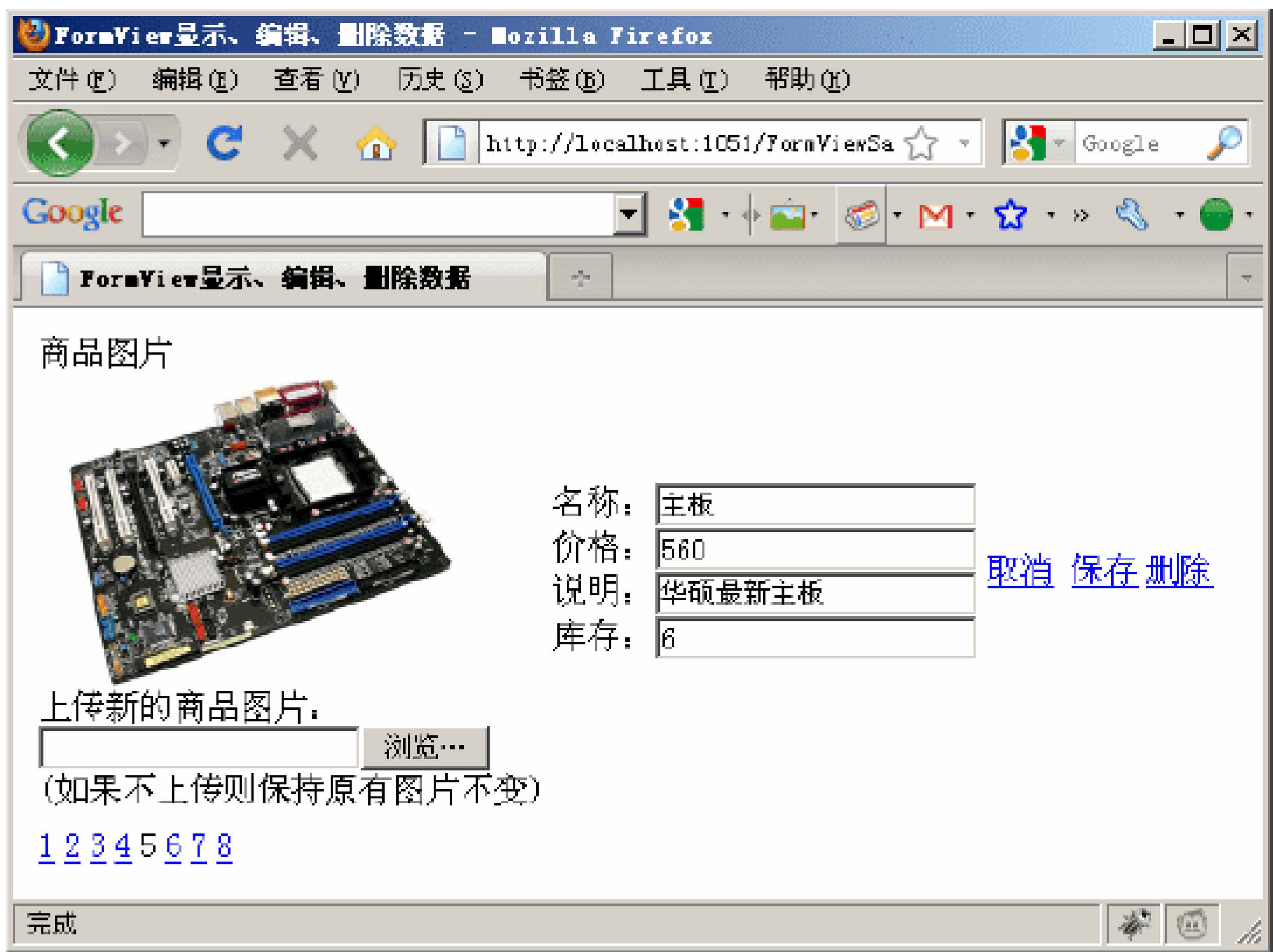


图 3.31　FormView 示例

3.5　数据源控件

数据源控件是管理连接到数据源及读取和写入数据等任务的 ASP.NET 控件。数据源控件不呈现任何用户界面，而是充当特定数据源（如数据库、业务对象或 XML 文件）与 ASP.NET 网页上的其他控件之间的中间方。数据源控件实现了丰富的数据检索和修改功能，其中包括查询、排序、分页、筛选、更新、删除以及插入。数据源控件封装了 ADO.NET 的功能，如连接、命令、数据读取等。

3.5.1　SqlDataSource 控件

SqlDataSource 控件使用 ADO.NET 类与 ADO.NET 支持的任何数据库进行交互。这类数据库包括 Microsoft SQL Server（使用 System.Data.SqlClient 提供程序）、System.Data.OleDb、System.Data.Odbc 和 Oracle（使用 System.Data.OracleClient 提供程序）。使用 SqlDataSource 控件，可以在 ASP.NET 页中访问和操作数据，而无须直接使用 ADO.NET 类。只需提供用于连接到数据库的连接字符串，并定义使用数据的 SQL 语句或存储过程即

可。在运行时，SqlDataSource 控件会自动打开数据库连接，执行 SQL 语句或存储过程，返回选定数据，然后关闭连接。

提示：SqlDataSource 类并非专门用于连接 SQLServer 数据库，而是可以连接各种 ADO.NET 支持的数据源。有人曲解了 SqlDataSource 数据源的名称，认为 SQL 就是 SQL Server，其实这里的 SQL 是泛指一般意义上的 SQL，而非特指微软公司的 SQL Server 数据库。

【例 3-18】 SqlDataSource 控件。

本例使用 SqlDataSource 作为数据源，GridView 作为显示控件，显示商品信息列表。

（1）创建一个 ASP.NET 应用程序，把包含 Products 表的数据库复制到 App_Data 目录下。然后在项目中添加一个新页面 SqlDataSourceSample.aspx。

（2）在页面上放置一个 SqlDataSource 控件。单击控件右上角智能按钮，打开智能任务面板，选择“配置数据源”，如图 3.32 所示。

（3）之后会弹出如图 3.33 所示的“配置数据源”对话框。

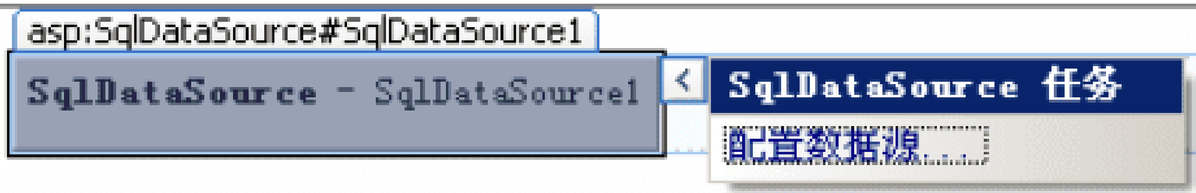


图 3.32　SqlDataSource 智能任务面板

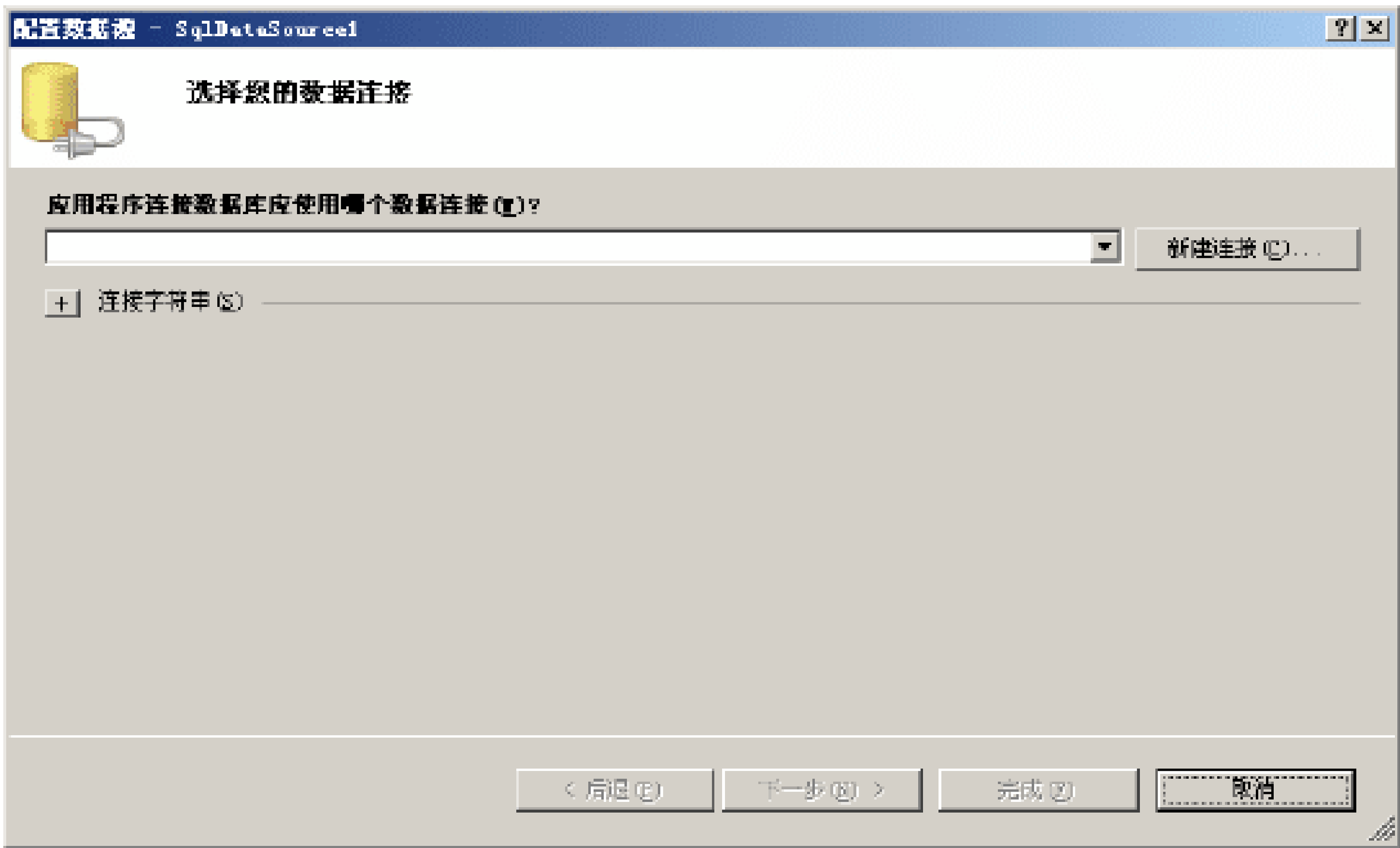


图 3.33　“配置数据源”对话框

（4）在图 3.33 所示对话框中单击“新建连接”按钮，则弹出如图 3.34 所示的“添加连接”对话框。

（5）在图 3.34 所示对话框中单击“更改”按钮，则弹出如图 3.35 所示的“更改数据源”对话框。在这个对话框中，列出了 SqlDataSource 所支持的各种数据源，如 ODBC、SQL Server、Access 等，可以选择需要连接的数据源类型。

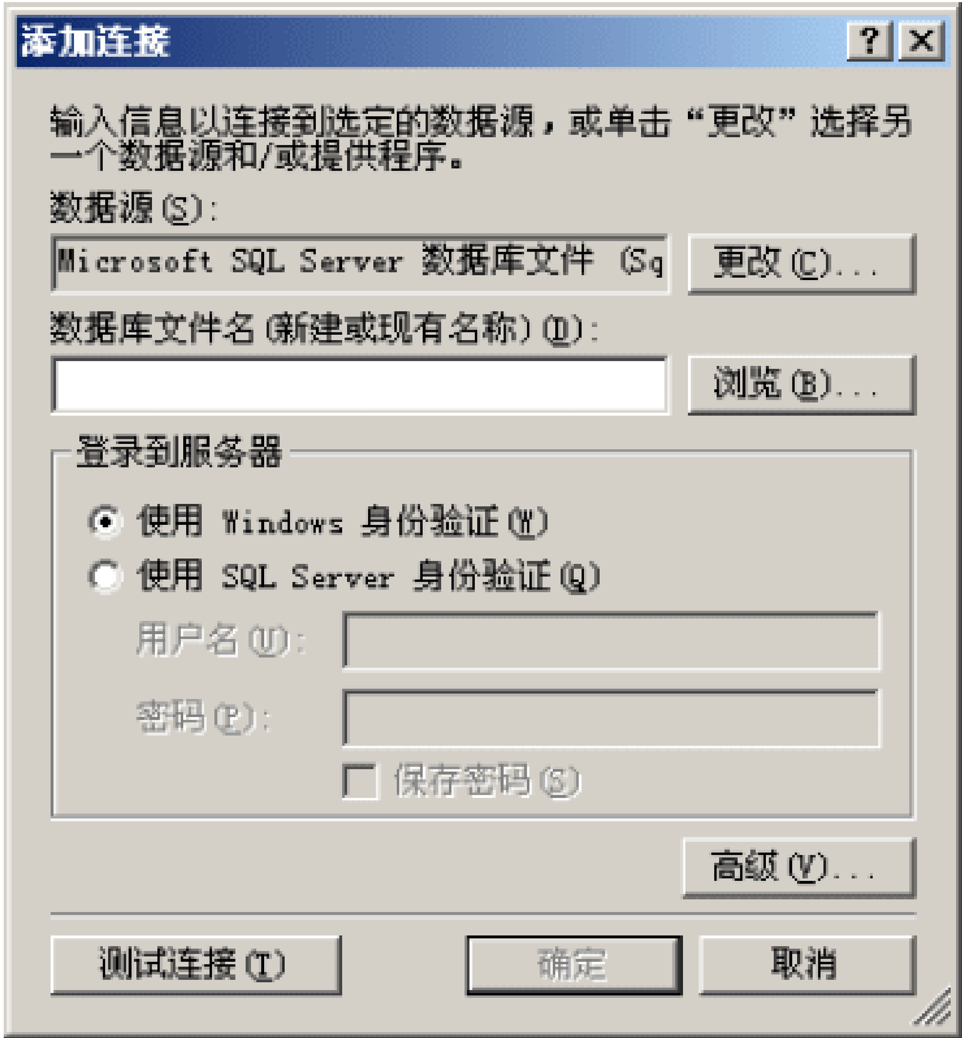


图 3.34　“添加连接”对话框

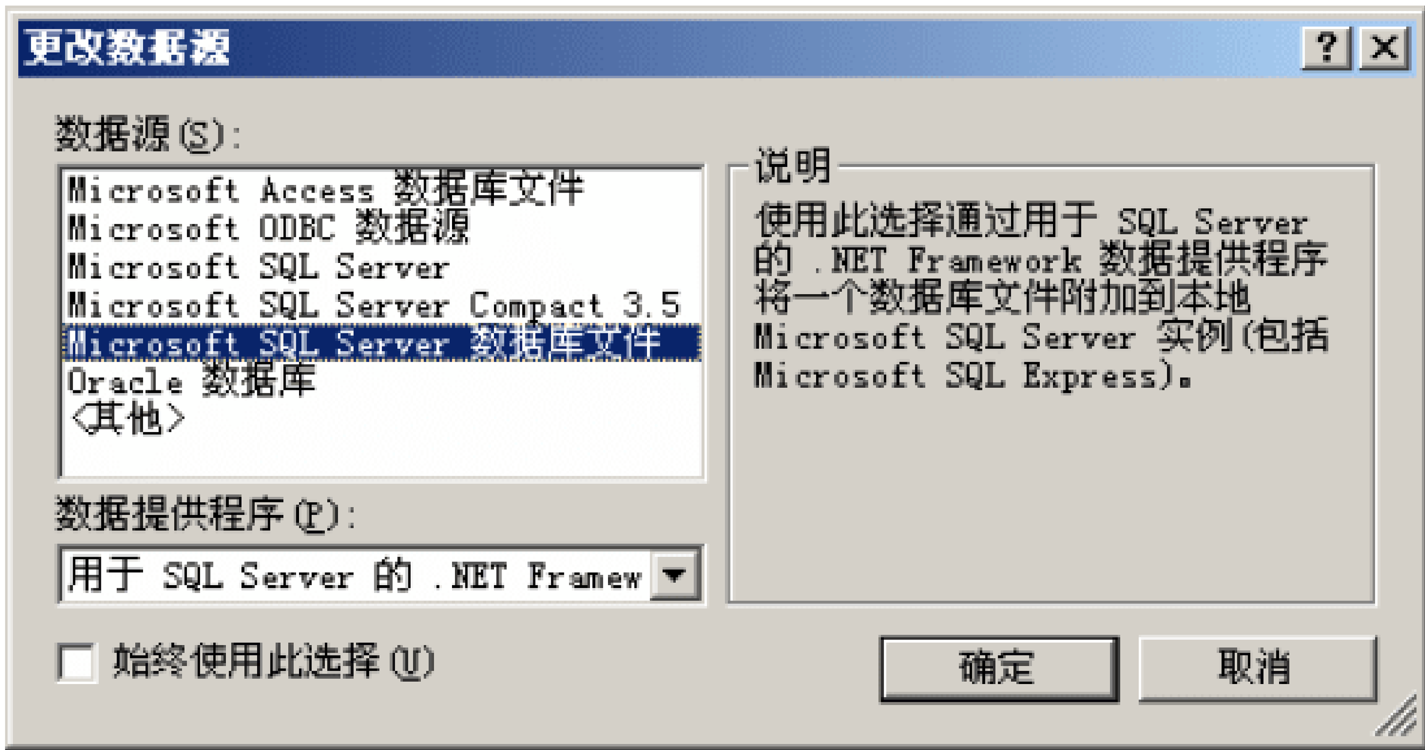


图 3.35　“更改数据源”对话框

（6）本例需要连接 App_Data 目录下的数据库文件，所以从图 3.35 所示对话框中选择“Microsoft SQL Server 数据库文件”，单击“确定”按钮，然后从弹出的对话框中选择项目中的数据库文件。接下来，会弹出“配置 Select 语句”对话框，如图 3.36 所示。在这个对话框中可以设置查询所需要的 SQL 语句。本例中，选择 Products 表的各个字段。

（7）SqlDataSource 控件支持自动生成编辑数据所需的 Insert、Delete 和 Update 语句，从而允许不编写代码即可实现数据的编辑、添加和删除。在图 3.36 中，单击“高级”按钮，则弹出如图 3.37 所示的“高级 SQL 生成选项”对话框，在其中选中“生成 INSERT、UPDATE 和 DELETE 语句”单选按钮。

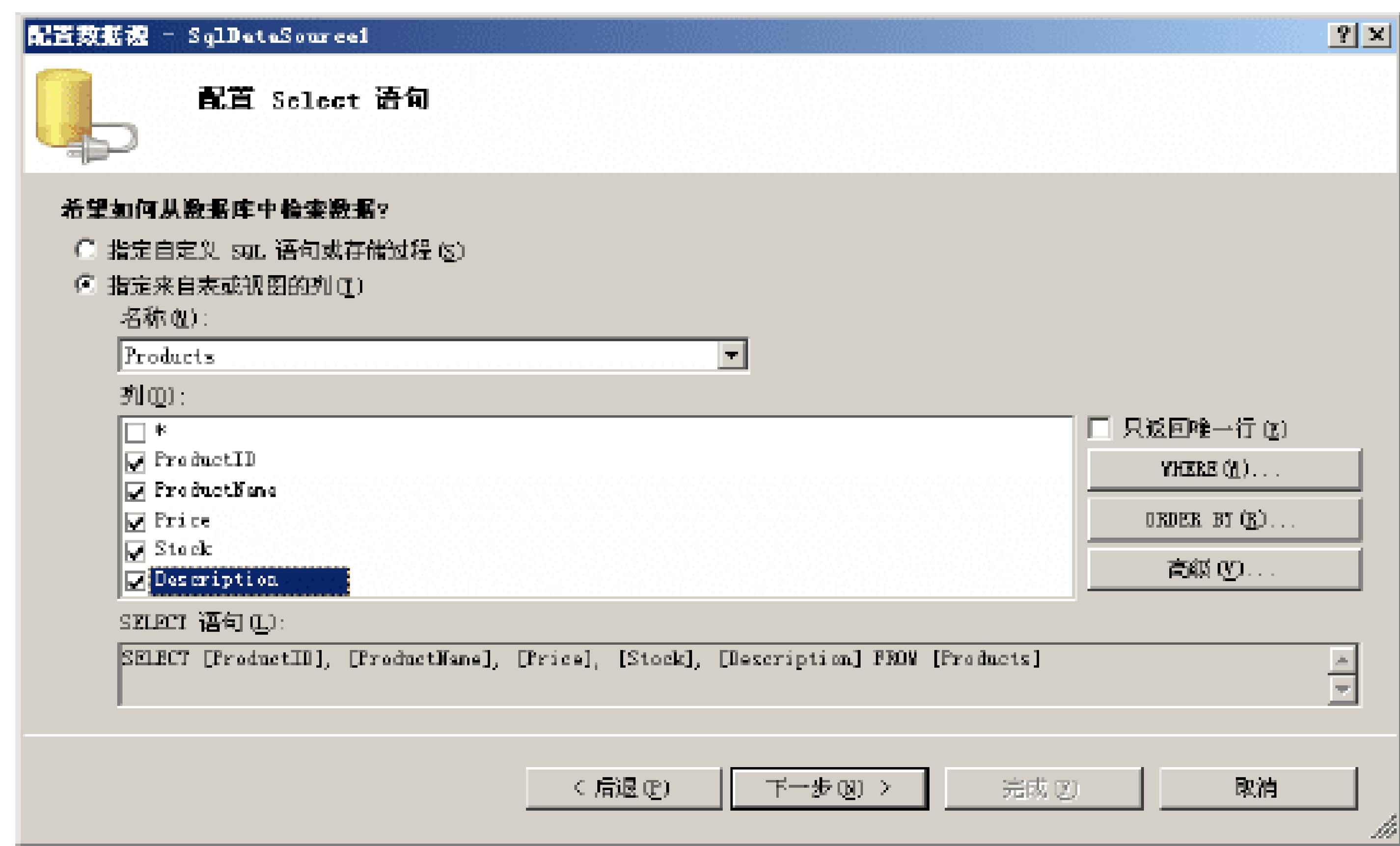


图 3.36　配置 Select 语句

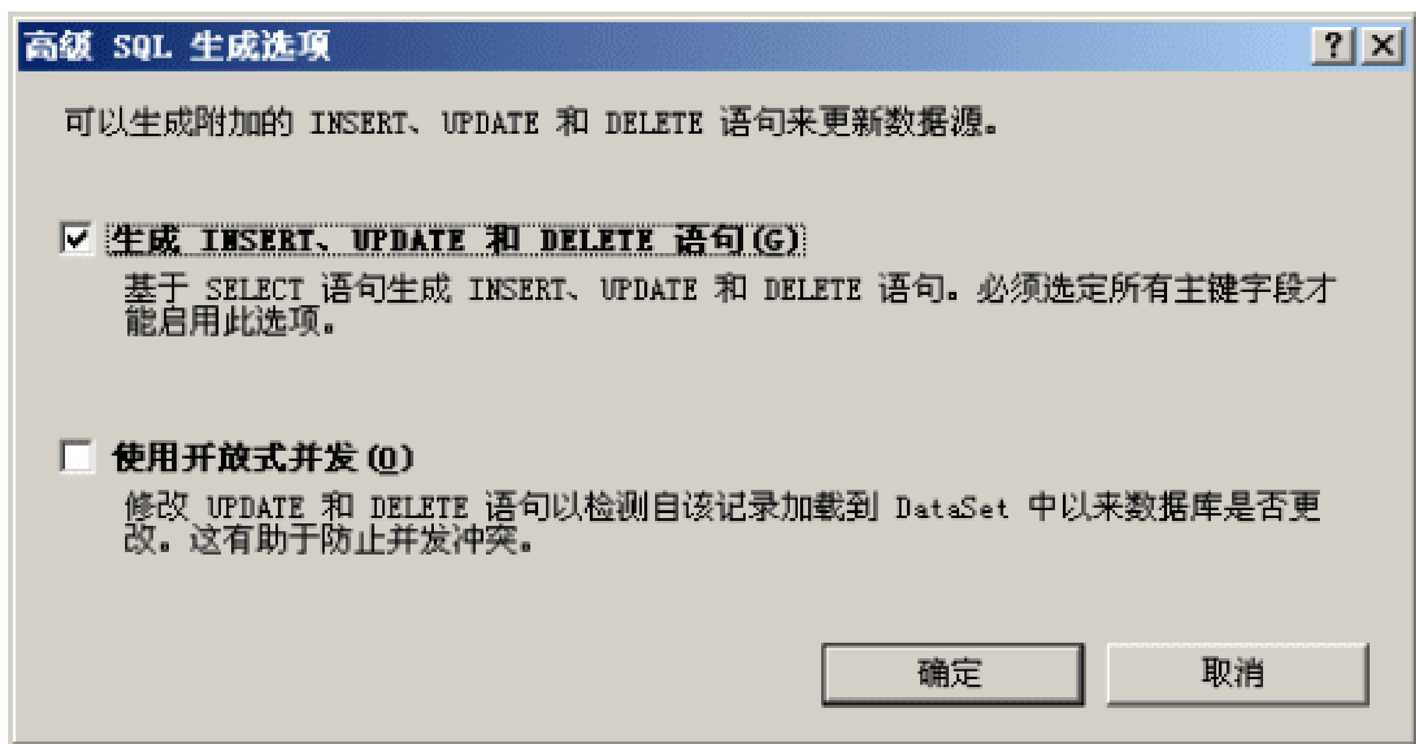


图 3.37　高级 SQL 生成选项

（8）在图 3.37 所示的对话框中单击“确定”按钮以后，回到图 3.36 所示对话框，一直单击“下一步”按钮，直到配置结束，单击“完成”按钮。配置结束以后，在页面中生成以下代码：

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
   ConnectionString="<%$ ConnectionStrings:database %>"
   DeleteCommand="DELETE FROM [Products] WHERE [ProductID] = @ProductID"
   InsertCommand="INSERT INTO [Products] ([ProductID], [ProductName],
   [Price], [Stock], [Description]) VALUES (@ProductID, @ProductName,
    @Price, @Stock, @Description)"
   SelectCommand="SELECT [ProductID], [ProductName], [Price], [Stock],
   [Description] FROM [Products]"
   UpdateCommand="UPDATE [Products] SET [ProductName] = @ProductName,
   [Price] = @Price, [Stock] = @Stock, [Description] = @Description WHERE
    [ProductID] = @ProductID">
   <DeleteParameters>
      <asp:Parameter Name="ProductID" Type="String" />
   </DeleteParameters>
   <InsertParameters>
      <asp:Parameter Name="ProductID" Type="String" />
      <asp:Parameter Name="ProductName" Type="String" />
      <asp:Parameter Name="Price" Type="Double" />
      <asp:Parameter Name="Stock" Type="Double" />
      <asp:Parameter Name="Description" Type="String" />
   </InsertParameters>
   <UpdateParameters>
      <asp:Parameter Name="ProductName" Type="String" />
      <asp:Parameter Name="Price" Type="Double" />
      <asp:Parameter Name="Stock" Type="Double" />
      <asp:Parameter Name="Description" Type="String" />
      <asp:Parameter Name="ProductID" Type="String" />
   </UpdateParameters>
</asp:SqlDataSource>
```

以上代码主要包括 3 部分内容，第 1 部分定义了连接字符串，第 2 部分定义了 Select、Insert、Update 和 Delete 语句，第 3 部分定义了这 4 条语句用到的参数。从这些代码内容来看，SqlDataSource 封装了数据库连接和增删改查 4 个命令，其作用相当于数据适配器 SqlDataAdapter。

（9）在页面上放置一个 GridView 控件，从其智能任务面板中选择 SqlDataSource1 作为数据源，如图 3.38 所示。

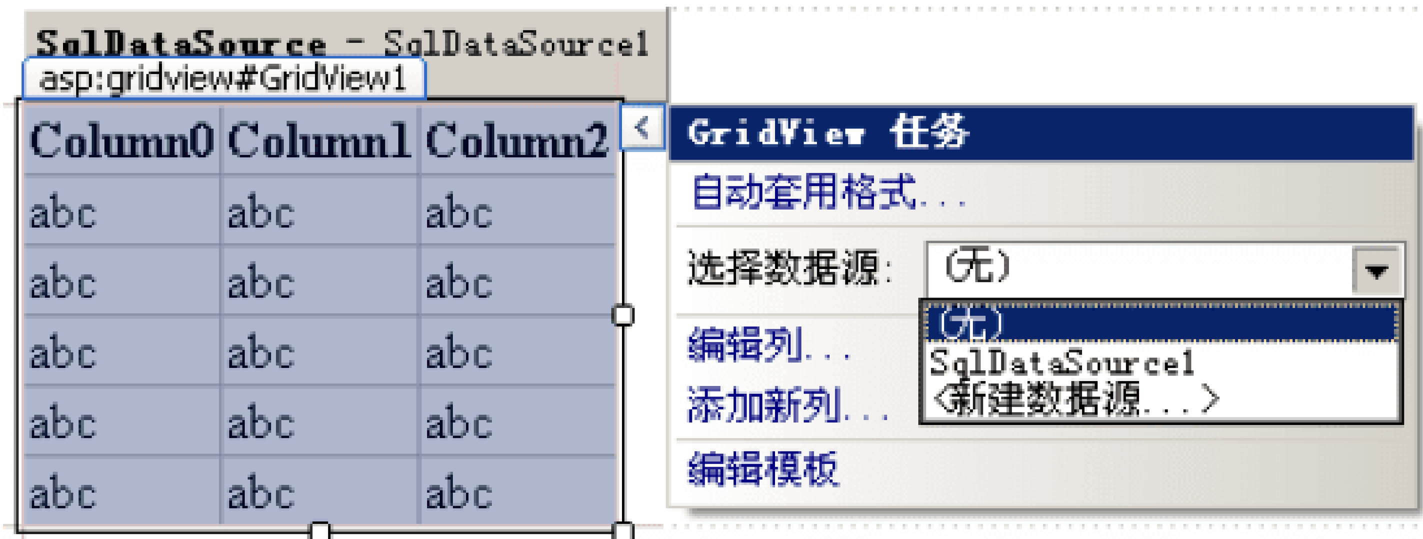


图 3.38　为 GridView 选择 SqlDataSouce 作为数据源

（10）为 GridView 选中 SqlDataSource1 作为数据源以后，GridView 会根据数据源中的字段自动生成列，而且 GridView 任务面板也多出几个选项，包括启用分页、启用编辑、启用删除等，如图 3.39 所示。

（11）为了演示 SqlDataSource 与 GridView 配合工作的功能，在图 3.39 所示对话框中，把启用分页、启用排序、启用编辑、启用删除和启用选定内容都选中。为 GridView 各个字段指定中文列标题，浏览 SqlDataSourceSample.aspx 页面，运行界面如图 3.40 所示。从运行结果可以看到，整个例子中没有手工编写一行代码，实现了数据的显示、分页、排序、删除、编辑功能。



图 3.39　设定了 SqlDataSource 数据源以后的 GridView

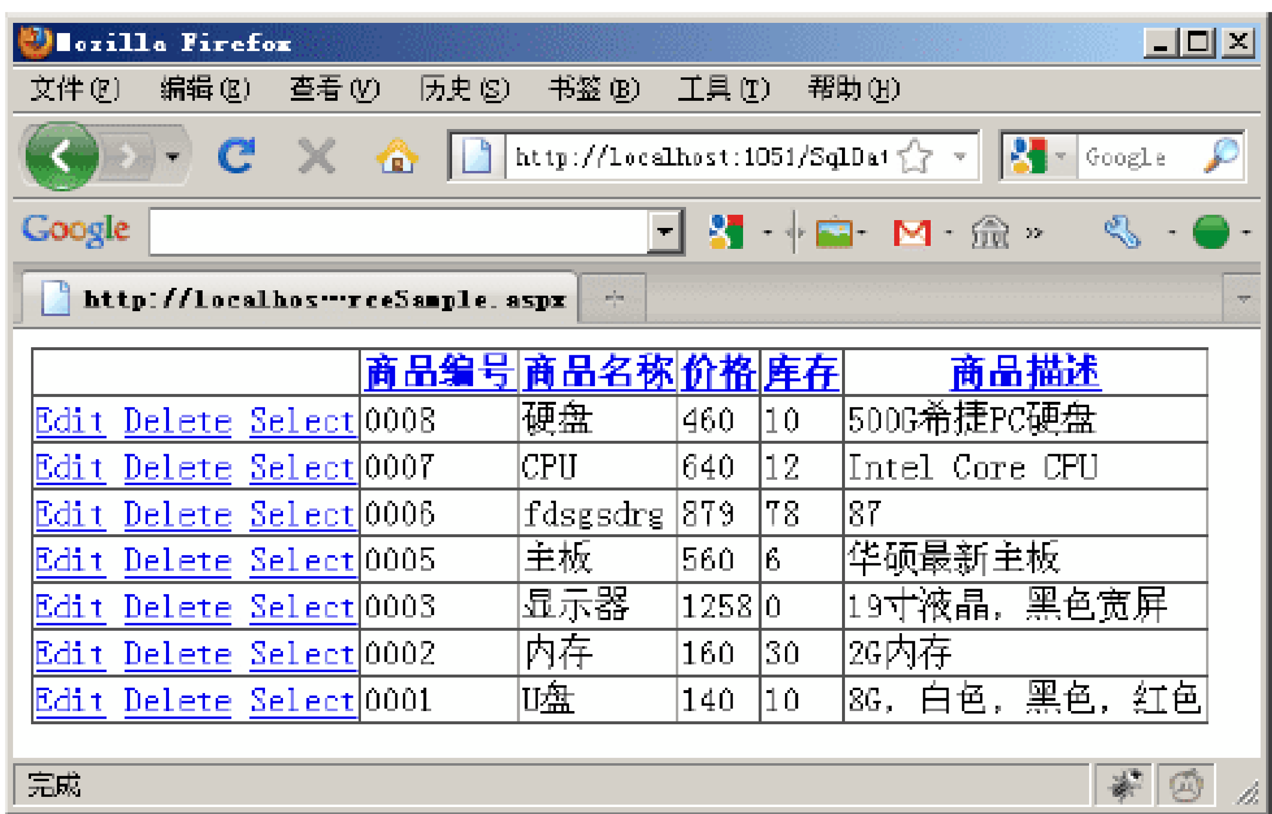


图 3.40　SqlDataSource 示例运行界面

3.5.2　数据源控件参数

在很多情况下，页面上并不需要显示所有数据，而需要根据用户的选择进行数据的查询和显示。例如，在网上书店程序中，用户可能通过输入关键字（如作者、书名等）或者选择图书类别进行查询，程序需要根据用户输入的条件执行相应的检索并显示数据。这种情况下生成的 SQL 查询语句以用户输入内容为参数。SqlDataSource 提供了灵活强大的参数化查询功能，可以实现这种需求。

【例 3-19】SqlDataSource 参数化查询。

本例通过 SqlDataSource 参数化查询实现用户搜索商品的功能。

（1）创建一个 ASP.NET 应用程序，把商品信息数据库和商品图片复制到项目中。

（2）在项目中添加一个页面 ParameterizedQuery.aspx，在页面上放置一个 SqlDataSource 控件和一个 GridView 控件，代码如下：

```
根据名称搜索：<asp:TextBox ID="productName" runat="server"></asp:TextBox>
<asp:Button ID="Button1"  runat="server" Text="搜索" />
<asp:GridView ID="GridView1" runat="server">
</asp:GridView>
<asp:SqlDataSource ID="SqlDataSource1" runat="server"></asp:SqlDataSource>
```

（3）为 SqlDataSource 设置数据源，大体步骤同例 3-18。与例 3-18 不同的是，当出现如图 3.41 所示的“配置 Select 语句”对话框时，选中 Products 表的各个字段后，要单击对话框右侧的 WHERE 按钮。

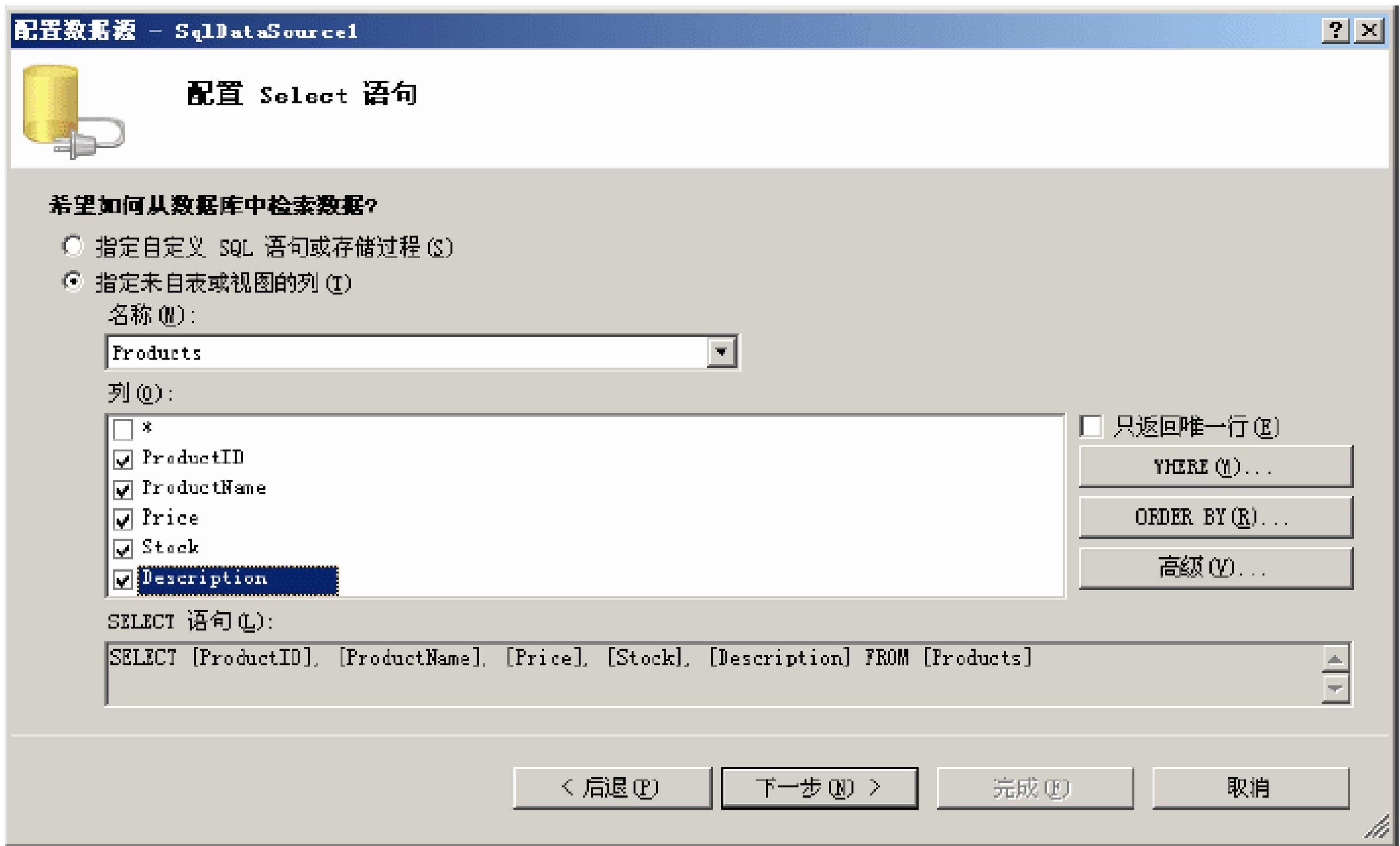


图 3.41　配置 Select 语句

（4）之后会弹出如图 3.42 所示的“添加 WHERE 子句”对话框。本例的功能是需要根据用户输入的产品名称进行搜索，参照图 3.42 在这个对话框中进行适当配置，然后单击“添加”按钮，则会将此查询条件添加到 SqlDataSource 中的 Select 语句中。

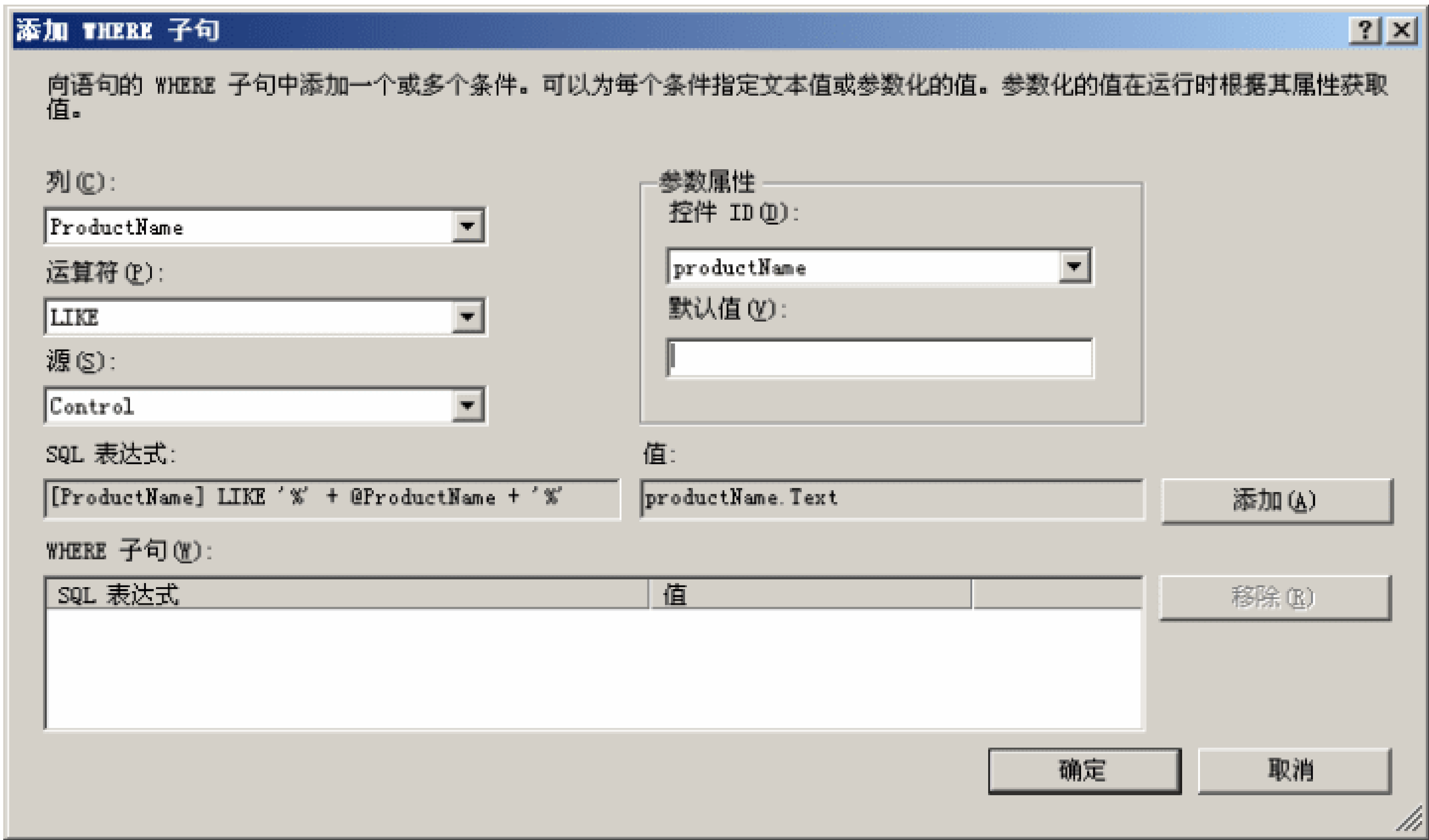


图 3.42　添加 Where 子句

（5）SqlDataSource 配置完成后，页面代码变成以下内容。

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%$ ConnectionStrings:database %>"
```

```
    SelectCommand="SELECT [ProductID], [ProductName], [Price], [Stock],
    [Description] FROM [Products] WHERE ([ProductName] LIKE '%' +
    @ProductName + '%')">
    <SelectParameters>
       <asp:ControlParameter ControlID="productName" Name="ProductName"
          PropertyName="Text" Type="String" />
    </SelectParameters>
</asp:SqlDataSource>
```

（6）将 GridView 控件的数据源设置为 SqlDataSource 控件，并且为各个字段添加标题。

```
<asp:GridView ID="GridView1" runat="server"  AutoGenerateColumns="False"
DataKeyNames="ProductID"
    DataSourceID="SqlDataSource1" PageSize="5" >
          <Columns>
       <asp:BoundField DataField="ProductID" HeaderText="商品编号" Read-
       Only="True" SortExpression="ProductID" />
       <asp:BoundField DataField="ProductName" HeaderText="商品名称"
          SortExpression="ProductName" />
       <asp:BoundField DataField="Price" HeaderText="价格" SortExpression=
       "Price" />
       <asp:BoundField DataField="Stock" HeaderText="库存" SortExpression=
       "Stock" />
       <asp:BoundField DataField="Description" HeaderText="商品描述"  Sort-
       Expression="Description" />
    </Columns>
</asp:GridView>
```

（7）运行页面，在搜索框中输入关键字并查询，页面运行效果如图 3.43 所示。

（8）本例到此为止实现了产品搜索功能，下一个步骤在 GridView 中添加一个链接，单击此链接打开一个新页面，显示商品详情，这也是大多数购物网站都具备的一个功能。在 GridView 控件中添加一个新的超链接列，以打开商品详情页面。在 GridView 控件的智能任务面板中选择“添加字段”，则打开“添加字段”对话框，参照图 3.44 设置各个选项。

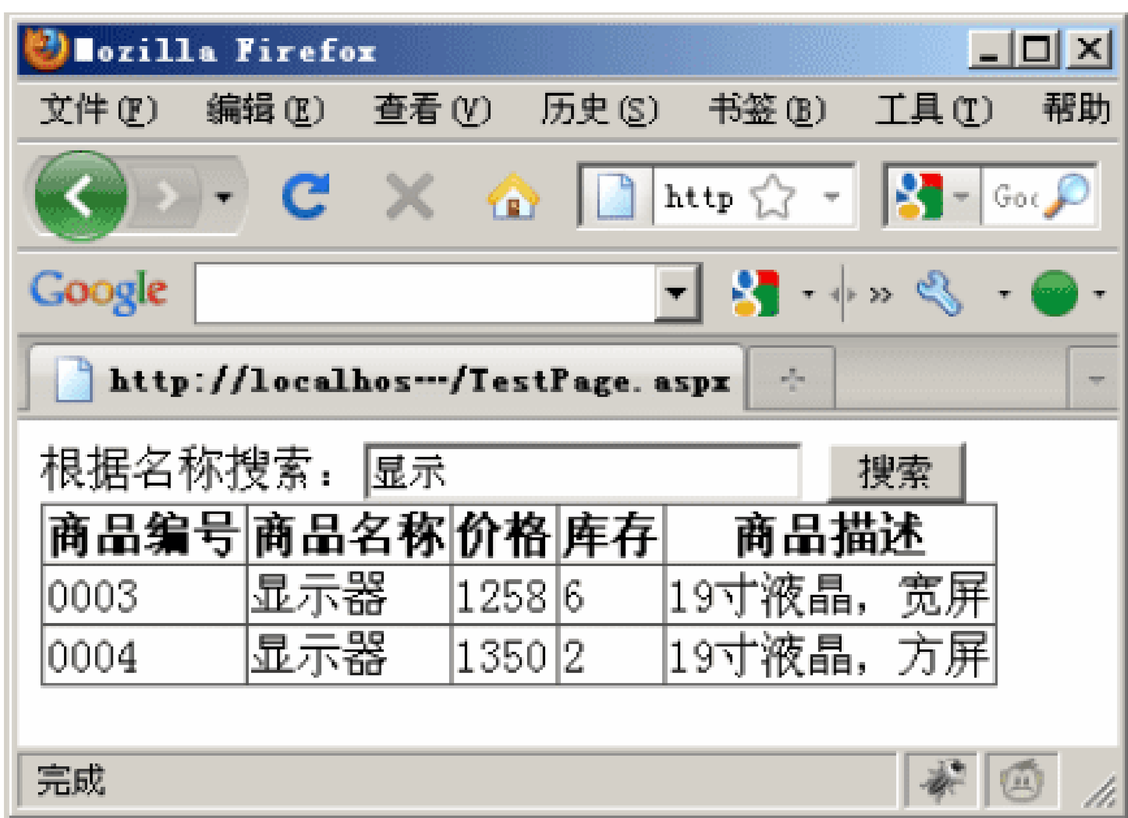


图 3.43　利用 SqlDataSource 参数查询商品

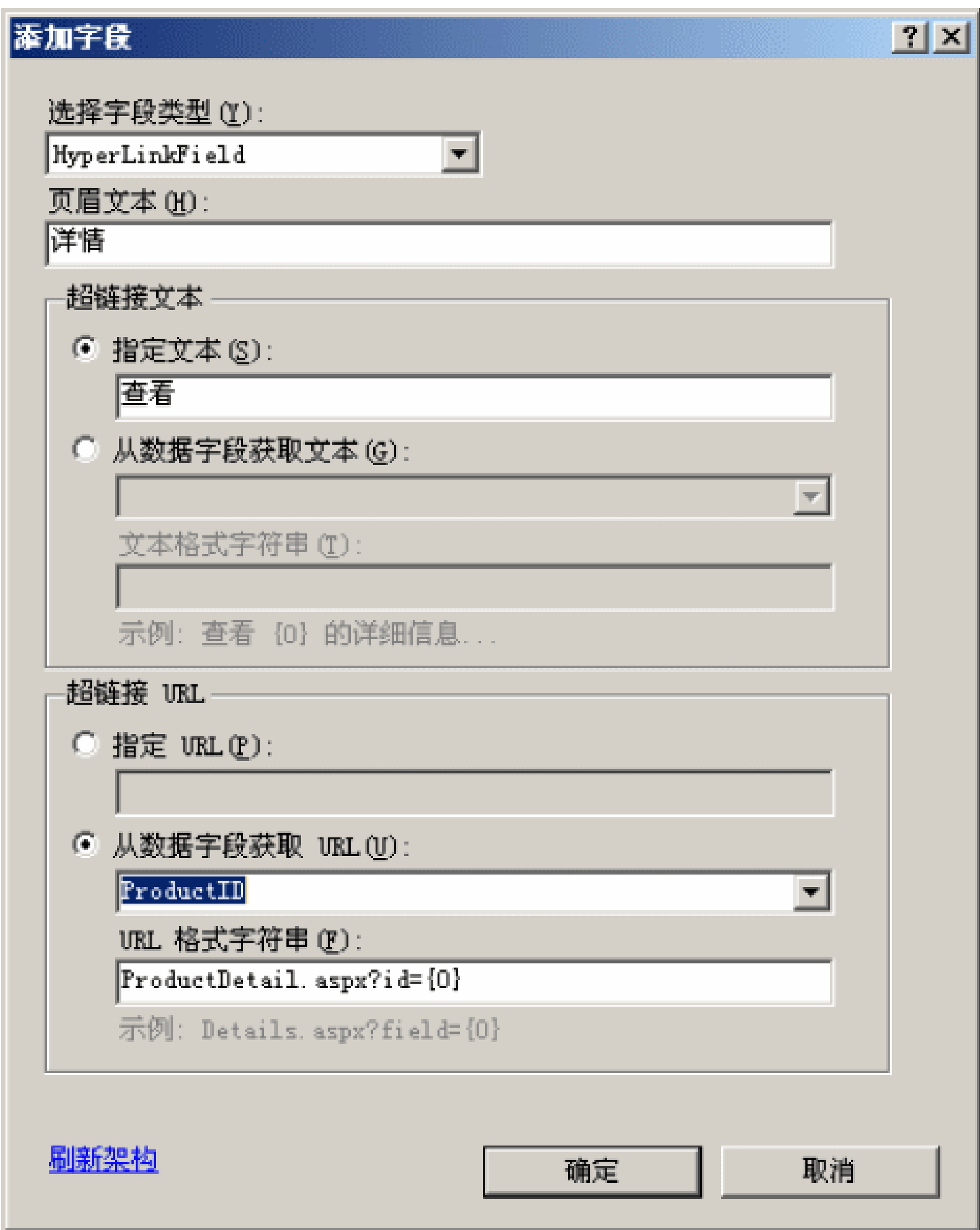


图 3.44　在 GridView 添加超链接字段

GridView 添加超链接字段后，生成如下代码：

```
<asp:HyperLinkField DataNavigateUrlFields="ProductID"
   DataNavigateUrlFormatString="ProductDetail.aspx?id={0}" HeaderText="
   详情"
   Text="查看" Target="_blank"  />
```

上述代码含义为，当在浏览器中单击此链接时，会在新窗口中打开 ProductDetail.aspx 页面，并将当前商品 ProductID 字段的值作为 QueryString 参数传递给新页面。

（9）在项目中添加一个新的页面 ProductDetail.aspx 以显示商品详情。在 ProductDetail.aspx 页面上放置一个 SqlDataSource 以从数据库检索数据。在配置 SqlData Srouce 时要注意添加一个 Where 条件，此条件的参数来源于 QueryString，具体配置界面如图 3.45 所示。

配置结束后，SqlDataSource 代码如下：

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
   ConnectionString="<%$ ConnectionStrings:database %>"
   SelectCommand="SELECT [ProductID], [ProductName], [Price], [Stock],
   [Description], [ImageUrl] FROM [Products] WHERE ([ProductID] =
   @ProductID)">
   <SelectParameters>
      <asp:QueryStringParameter Name="ProductID" QueryStringField="id"
         Type="String" />
   </SelectParameters>
</asp:SqlDataSource>
```

（10）在 ProductDetail.aspx 页面上放一个 FormView 以显示商品详情。代码如下：

```
<asp:FormView ID="FormView1" runat="server" DataSourceID="SqlDataSource1" >
   <ItemTemplate>
   <table>
      <tr>
      <td>  <img  src="ProductImages/<%#Eval("ImageUrl") %>" alt="<%#Eval
      ("ProductName") %>" style="width:200px;" /></td>
      <td>
      <span style="font-weight:bold;"><%#Eval("ProductName")%></span><br />
       价格：<%#Eval("Price","{0:C}") %><br />说明：<%#Eval("Description")
       %><br />库存：<%#Eval("Stock") %></td>
      </tr>
   </table>
   </ItemTemplate>
</asp:FormView>
```

（11）运行 ParameterizedQuery.aspx 页面，并单击一个商品的“查看”链接，打开 Product Detail.aspx 页面并显示该商品详细信息，如图 3.46 所示。

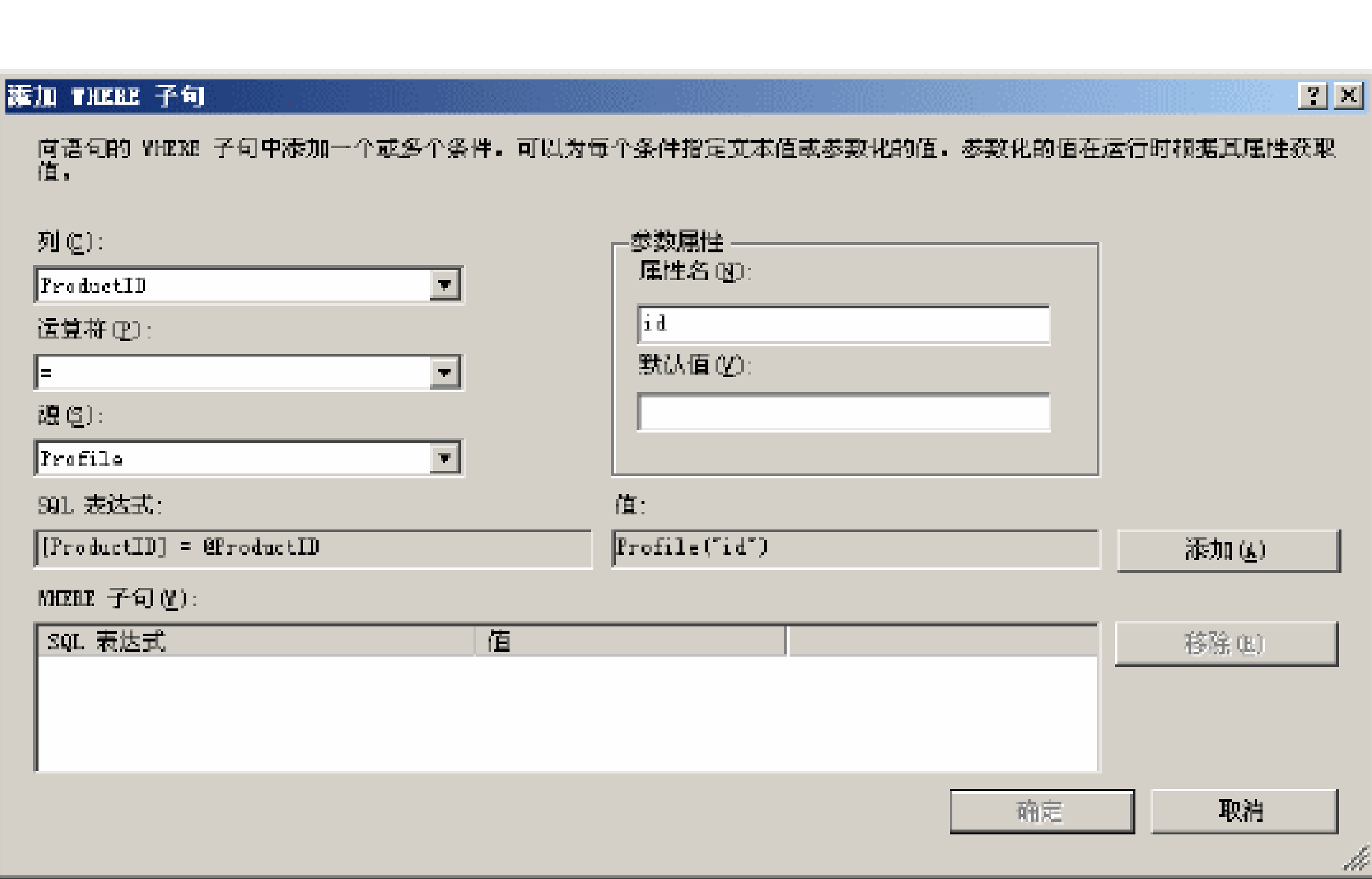


图 3.45　为 SqlDataSource 添加 QueryString 参数

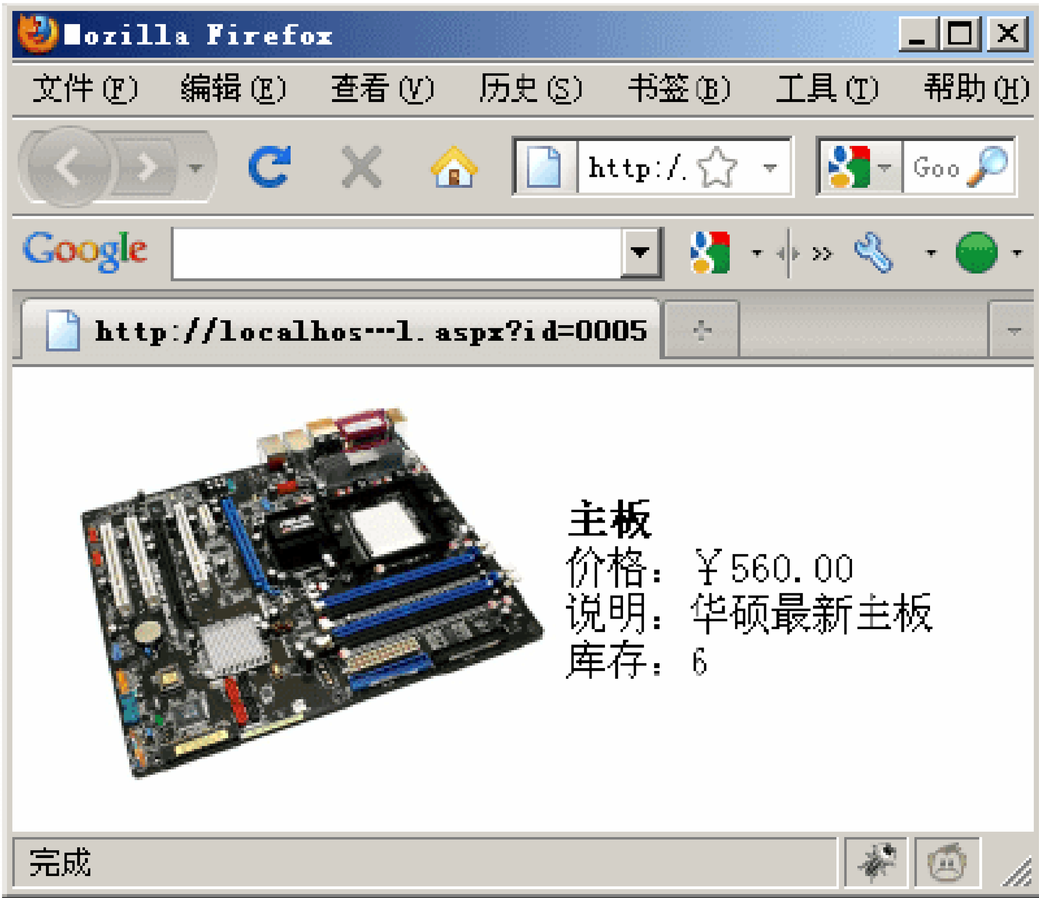


图 3.46　根据 QueryString 传参显示商品详情页面

3.5.3　其他数据源控件

除 SqlDataSource 数据源控件以外，ASP.NET 还主要包括以下数据源控件。

（1）EntityDataSource：用于连接到实体框架 Entity Framework 数据源。

（2）AccessDataSource：连接到 Microsoft Access 数据库。当数据作为 DataSet 对象返回时，支持排序、筛选和分页。

（3）XmlDataSource：连接到 XML 文件。

（4）SiteMapDataSource：ASP.NET 站点地图数据源控件。

（5）LinqDataSource：支持语言集成查询 LINQ。支持自动生成选择、更新、插入和删除命令。该控件还支持排序、筛选和分页。LINQ 将在本书后续章节中介绍。

（6）EntityDataSource：支持实体框架数据源。支持自动生成更新、插入、删除和选择命令。该控件还支持排序、筛选和分页。实体框架 Entity Framework 将在本书后续章节中介绍。

（7）ObjectDataSource：允许使用业务对象或其他类，以及创建依赖中间层对象管理数据的 Web 应用程序。支持对其他数据源控件不可用的高级排序和分页方案。关于多层结构的相关知识将在本书后续内容中介绍。

这些数据源控件的使用与 SqlDataSource 控件大同小异，不再一一详细介绍。

3.6　小　　结

本章介绍了 ASP.NET 数据控件。ASP.NET 数据控件主要包括用于显示数据的数据绑定控件（如 GridView、DataList、DetailsView 等）和用于连接数据源的数据源控件（如 SqlDataSource、AccessDataSource、LinqDataSource 等）。这些数据控件封装了常用的数据显示和数据访问功能，提高了开发效率。这些控件在各种 ASP.NET 数据库应用程序中广泛使用。

第 4 章　阶段项目案例：网上书店

在学习软件开发技术过程中，很多人都会遇到一个问题：技术细节学了一堆，但是却不能融会贯通，不能从整体上把握这些具体技术细节在框架中的地位及互相关系，不能将各种技术细节综合起来应用到实际项目中。为了避免出现这种“只见树木，不见森林”的情况，本章将综合利用前面几章学习过的 ASP.NET 编程和数据访问相关技术开发一个网上书店，以达到技术整合和项目锻炼的目的。

4.1　网上书店整体设计

在具体编码开发网上书店项目以前，先要分析此项目的需求，并设计出网站的整体结构。为了叙述方便，将本章所开发的网上书店称为“攀登者网上书店”。本节将分析这个网上书店的主要功能需求，并设计出数据库结构。

4.1.1　功能需求

攀登者网上书店是一个典型的网上书店，功能与其他书店基本相同。网站功能整体上可以分为两大部分：前台功能和后台功能。网站前台应该包括以下功能。

（1）图书信息展示和浏览功能。网站能够显示图书列表，从图书列表可以导航到图书详细信息。

（2）图书搜索功能。用户可以根据某些条件（如书名、作者、出版社等）搜索图书，并将搜索结果按照一定顺序排序（如价格、销售量等）。

（3）购物车功能。用户在浏览图书过程中，随时可以把感兴趣的图书放到购物车，并可随时查看和修改购物车中存放的图书。

（4）会员功能。用户可以在网上注册为会员，可以填写并修改会员相关信息（如联系电话、送货地址等）。

（5）订单功能。用户可以下订单购买图书，并在一定条件下（通常是订单尚未处理时）可以修改和取消订单。

（6）评论功能。用户可以对图书进行评论。

攀登者网上书店后台主要包括以下功能。

（1）登录和用户权限检测。只有管理中才允许进入后台管理页面，所以后台管理模块需要检测当前用户是否拥有足够的权限。

（2）图书类别管理。浏览、添加、删除、修改图书类型信息。

（3）图书管理。浏览、添加、删除、修改图书信息，改变图书分类等。

（4）会员管理。可以浏览、添加、删除、修改会员信息。

（5）订单管理。可以查询订单，改变订单状态（如将订单状态修改为已发货），删除订单。

（6）评论管理。可以查看会员对图书的评论，并可以有选择的删除这些评论。

提示：限于篇幅，本章只给出了较为典型和重要且不重复的代码，主要包括图书列表显示、图书搜索、购物车、身份验证、图书信息维护。对于定单和评论功能，其实现思路与前面这些功能类似，而且代码比前者更少，本章没有给出具体实现。

4.1.2　数据库结构设计

根据前面对攀登者网上书店功能分析，可以设计出网站的数据库结构。网站主要包括以下几个表。

（1）类别表 Category。

- ❑ CategoryID：类别 ID，int 类型，主键，自动增长列。
- ❑ CategoryName：类别名称，nvarchar(20)类型。

（2）图书表 Book。

- ❑ BookID：图书 ID，int 类型，主键，自动增长列。
- ❑ BookTitle：图书标题，nvarchar(50)。
- ❑ Author1、Author2、Author3：三列都是 nvarchar(10)类型，分别表示第一作者、第二作者和第三作者。
- ❑ PublishID：出版社 ID，int 类型，外键关联到 Pulisher 表。
- ❑ PublishDate：出版日期，Datetime 类型。
- ❑ Price：图书定价，float 类型。
- ❑ Discount：折扣比例，float 类型。
- ❑ Description：图书描述，nvarchar(1000)类型。
- ❑ Contents：图书目录，text 类型。
- ❑ SaleSum：该书总销售量，int 类型。
- ❑ ClickCount：该书总点击量，int 类型。
- ❑ SmallImage 和 BigImage：图书的大小图片。

（3）图书所属类别表 BookCategory。

- ❑ BookID：图书 ID，int 类型，主键，外键关联到 Book 表。
- ❑ CategoryID：类别 ID，int 类型，主键，外键关联到 Category 表。

提示：如果一本书只能属于一个类别，那么可以在图书表后面添加一个类别字段以标识图书所属的类别。但是，有的图书可以同时属于多个类别，这种情况就不能用前面的方法了，而是需要使用一个单独的表来描述图书和类别之间的多对多关系。

（4）会员表 Member。

- ❑ MemberID：会员 ID，int 类型，主键，自动增长列。

- MemberName：会员名称，nvarchar(20)类型，唯一性索引约束。
- Password：会员登录密码，nvarchar(20)类型。
- Address：送货地址，nvarchar(100)类型。
- Phone：联系电话，nvarchar(20)类型。
- Email：常用电子邮箱，nvarchar(20)类型。

（5）订单表 BookOrder。

- OrderID：订单 ID，int 类型，主键，自动增长列。
- MemberID：下订单的会员，int 类型，外键关联到 Member 表。
- OrderDate：下订单日期，Datetime 类型，默认为当前时间。
- ProcessFlog：订单是否已经处理，bit 类型，0 表示尚未处理，1 表示已经处理（如配货发货等）。

（6）订单明细表 OrderDetail。

- OrderID：订单编号，int 类型，外键关联到 Order 表。OrderID 与 BookID 组成双主键。
- BookID：订单所订图书 ID，int 类型，外键关联到 Book 表。OrderID 与 BookID 组成双主键。
- Price：图书价格，float 类型。
- BuyNumber：购买数量，int 类型。
- TotalMoney：总金额，等于 Price*BuyNumber。

（7）图书评论表 BookRemark。

- BookID：图书 ID，int 类型，外键关联到 Book 表。
- MemberID：会员 ID，int 类型，外键关联到 Member 表。
- Score：图书评分，int 类型（1～5 之间）。
- Remark：评论内容，nvarchar(500)类型。
- RemarkDate：评论时间，datetime 类型。

4.1.3　网站整体结构

攀登者网上书店是一个简单而典型的购物网站，拥有购物网站的主要模块，但规模不大，功能不复杂，适合作为练习项目。网站总体可以划分为两大组成部分：网站前台和网站后台。根据对网站功能需求的分析，攀登者网上书店网站整体结构如图 4.1 所示。

4.2　网上图书前台功能实现

攀登者网上书店总体包括前台和后台两大模块，这两大模块之间互相依赖，联系密切。例如，只有通过后台增加了图书信息，才能够在前台页面搜索和显示这些图书信息；只有在前台下了订单，后台才能够对订单进行处理。本节将介绍前台功能的实现，在 4.3 节再介绍后台功能的实现。

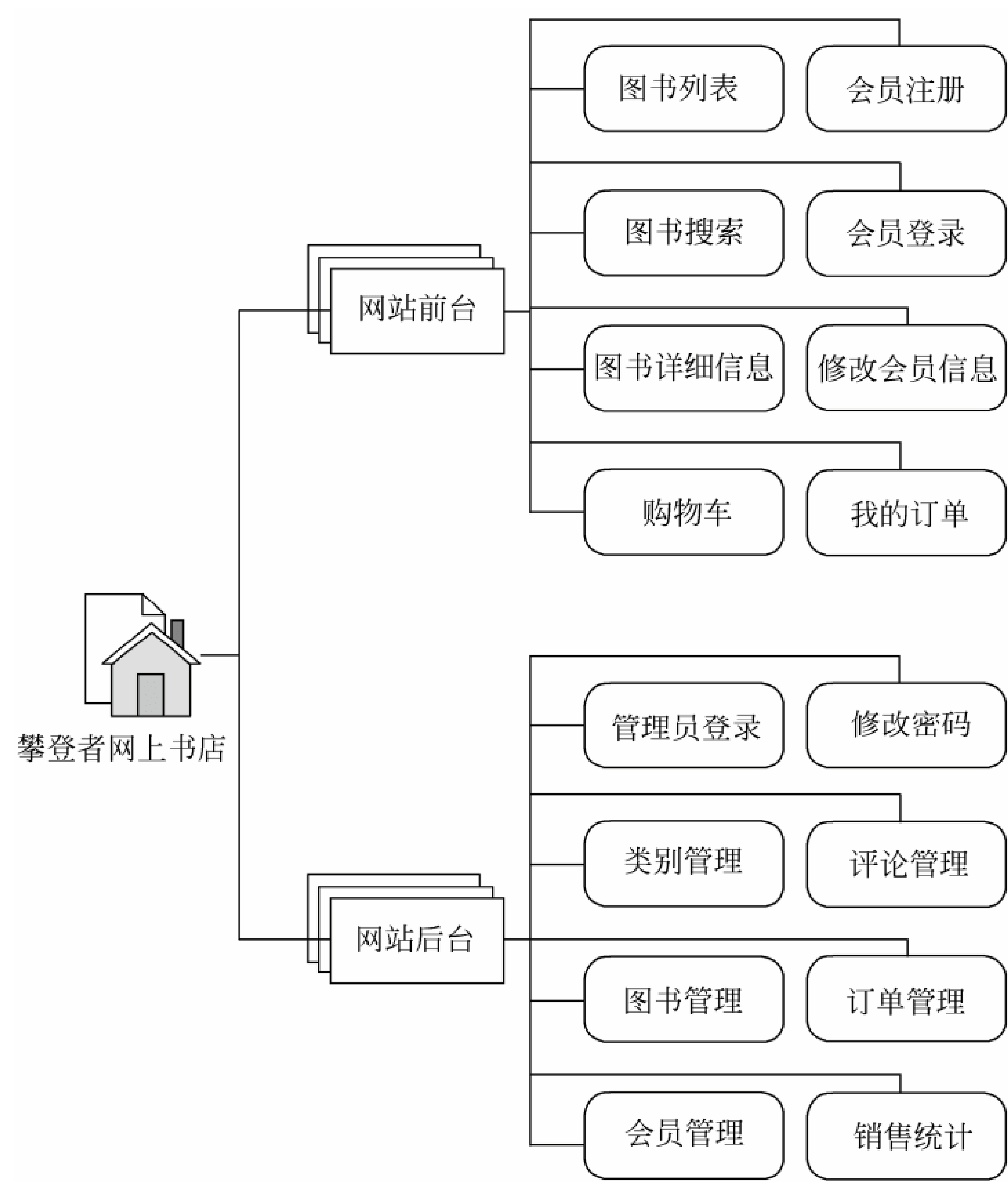


图 4.1　网上书店整体结构

4.2.1　母版页和主题设计

参照目前流行的网上书店页面结构，攀登者书店页面包括 3 大部分：页面顶部的 LOGO、广告和顶部导航栏、页面中间主要内容、页面底部的网站信息。其中页面顶部内容和页面底部内容对于大多数页面来说是相同的，可以把这些内容放到母版页中。母版页 BookShop.master 的效果如图 4.2 所示。



图 4.2　网上书店前台母版页

母版页 BookShop.master 代码如下：

```
<head runat="server">
    <title>攀登者网上书店</title>
    <link href="styles/bookshop.css" rel="stylesheet" type="text/css" />
    <asp:ContentPlaceHolder ID="head" runat="server">
    </asp:ContentPlaceHolder>
        <script type="text/javascript" language="javascript">
            //单击搜索按钮时调用此函数
            function simpleSearch() {
                //检查是否输入了检索关键字
                var key = document.getElementById("keyWord").value;
                if (!key) {
                    alert('请输入关键字再搜索');
                    return;
                }
                //页面跳转到简单搜索 SimpleSearch.aspx，并通过检索字符串传递参数
                window.location = "SimpleSearch.aspx?key=" + key;
            }
    </script>
</head>
<body>
<form id="form1" runat="server">
<div id="page">                                             <%--最外层 div--%>
<div id="header">                                           <%--页面头部--%>
<%--以下为 logo，广告图片和导航链接--%>
<img src="images/logo.jpeg" alt="攀登者网上书店" style="height:80px;" />
<img alt="大学优惠"
  src="images/banner.jpg" style="height:80px;" />
  <a href="ShopCartPage.aspx">
   <img src="images/cart.gif" alt="查看购物车" /></a> <br />
  <div style="padding:5px 0px; background-color:#e0f0ff; border:1px solid
green;">
  <a href="Default.aspx">本站首页</a> |
      <a href="#">销售排行</a> |
      <a href="ShopCartPage.aspx">购物车</a> |
  <a href="#">会员登录</a> |
  <a href="#">我的订单</a> |
  <a href="#">修改密码</a> |
  <a href="#">退出登录</a> |
  <%--以下为搜索框--%>
    搜索商品：<input type="text" id="keyWord" />
  <a href="javascript:simpleSearch();"><img
  src="images/search.jpg" style="vertical-align:middle;" alt="搜索"/></a>
  <a style="text-decoration:underline;" href="AdvanceSearch.aspx">高级搜索
</a>
  </div>
</div>                                                      <%--div header 结束--%>
<div style="clear:both;">                                   <%--页面主体--%>
    <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
    </asp:ContentPlaceHolder>
</div>
<hr style="color:silver;" />
<%--以下为页脚，包含链接、说明文字、图片、网站备案等内容--%>
<div id="footer">
<div style="float:left;">
    <img src="images/logo.jpeg" alt="" />
```

```
</div>
<div style="margin-left: 10px; float:left;" >
<a  href="Default.aspx">
<b>首页</b></a>   |  
<a href="#"><b>关于我们</b></a>    |  
 <a href="#"><b>诚聘英才</b></a>
 <sup><img src="images/new2.gif"></sup>|  
 <a href="#"><b>帮助中心</b></a>      |  
 <a  href="mailto:sun.j.l.studio@gmail.com"><b> 联 系 我 们 </b></a> 
 |  
 <a href="Default.aspx"><b>图书目录</b></a> <br/> <br />
 客户服务/电话订购: (010)88888888<br/>
 <span class="gray12"><a target="_blank" href="#">在线客服</a>
 <a target="_blank" href="#">联系客服</a></span>
 <br/><span class="orange13">工作时间 周一至周五 7:30-19:30 周六至周日
9:00-17:30</span><br />
 Copyright 2010 攀登者网上书店.All rights reserved </div>
<div style="margin: 20px 30pt 20px auto; float: right;">
<a target="_blank" href="http://www.315online.com. cn/member/315090031.
html">
<img border="0" align="left" title="网上交易保障中心" style="margin: 10px 10px
0pt 0pt;"
 src="images/bz315.gif"/></a>
   <div style="margin-top: 10px; float:left; "  >
   <a target="_blank" href="http://www.315online.com.cn/member/315090031.
   html">网上交易<br/>保障中心</a>
   </div>
   <div style="float:left;">
   <a target="_blank" href="http://www.hd315.gov.cn/beian/ view.
   asp?bianhao=010202001072000126">
   <img border="0" title="备案" src="images/biaoshi.gif"/></a> </div>
   <div style="float:left; margin-top:10px;">
   <a target="_blank" href="http://www.miibeian.gov.cn/">京 ICP 备<br
   />00011 号</a>
   </div></div>
</div>
</div>
</form>
</body>
</html>
```

为了给网站中主要控件添加统一美观的外观效果，在项目中添加一个主题 default，并在主题中添加一个外观文件，为 GridView 控件、TextBox 控件和 DropDownList 控件定义外观，代码如下：

```
<asp:GridView runat="server" CellPadding="4"  ForeColor="#333333"
GridLines="None" >
       <RowStyle CssClass="oddRow" />
       <FooterStyle BackColor="#507CD1" Font-Bold="True" ForeColor="White" />
       <PagerStyle BackColor="#2461BF" ForeColor="White" HorizontalAlign
       ="Center" />
       <SelectedRowStyle BackColor="#D1DDF1" Font-Bold="True" ForeColor
       ="#333333" />
       <HeaderStyle BackColor="#507CD1" Font-Bold="True" ForeColor="White"
       />
       <EditRowStyle BackColor="#2461BF" />
       <AlternatingRowStyle  CssClass="evenRow" />
</asp:GridView>
```

```
<asp:TextBox runat="server" CssClass="text"></asp:TextBox>
<asp:DropDownList runat="server" CssClass="select"></asp:DropDownList>
```

上述外观文件代码用到了如下几个 CSS 类。

```
.oddRow{ background-color:#EFF3FB;}                        //表格奇数行样式
.evenRow{background-color:White;}                          //表格偶数行样式
.hoverRow{background-color:#ffffdd;}                       //表格鼠标划过时样式
.text{ border:solid 1px #9af;}                             //文本框样式
.select{border:solid 1px; background-color:#ffd; width:120px;}//下拉框样式
```

在 web.config 文件中启用 default 主题。

```
<system.web>
   <pages styleSheetTheme="default"/>
</system.web>
```

4.2.2　网站中的通用类

在整个网上书店的开发过程中，会用到一些共同的功能，按照面向对象的设计思想，把这些功能封装成类，不但能够实现代码复用，而且使代码结构更加清晰。网站中用到的类主要包括图书类 Book、图书数据访问类 BookDAL、购物车项目类 ShopCartItem、购物车类 ShopCart 和常量容器类 Constants。

1. 图书类Book

Book 类描述了图书信息，其中包含的属性与数据库里的字段基本是一一对应的。

```
public class Book
{
   public int id { get; set; }                          //图书 ID
   public string title { get; set; }                    //图书标题
   public double price { get; set; }                    //单价
   public double discount { get; set; }                 //会员折扣
   public double realPrice { get; set; }                //打折后价格
   public string author1 { get; set; }                  //作者 1
   public string author2 { get; set; }                  //作者 2
   public string author3 { get; set; }                  //作者 3
   public string smallImage { get; set; }               //小封面图片
   public string bigImage { get; set; }                 //大封面图片
   public string content { get; set; }                  //内容简介
   public string description { get; set; }              //描述
   public int clickCount { get; set; }                  //点击数量
   public int saleSum { get; set; }                     //总销售量
   public int publisherID { get; set; }                 //出版社 ID
   public string publisherName { get; set; }            //出版社名称
   public DateTime publishDate { get; set; }            //出版日期
}
```

2. 图书数据访问类BookDAL

BookDAL 类实现了对于图书相关的数据库操作，如获取某一特定图书信息、按照条件

搜索图书等功能。由于类中包含的都是方法，而没有实体属性，所以把此类声明成静态类（static class），类中所有方法都是静态方法，从而允许通过类名直接调用方法而不需要实例化类，方便了方法的调用。BookDAL 类的第一个功能是根据图书编号获得图书信息。

```
/// <summary>
/// 根据图书 ID 从数据库读取图书信息
/// </summary>
/// <param name="bookid">要查询的图书 ID</param>
/// <returns>从数据库读取到的图书信息</returns>
public static Book getBookByID(int bookid)
{
    SqlHelper db = new SqlHelper();
    //构建查询用的 select 语句
    string sql = "select BookTitle,Author1,Author2,Author3, "
        +" Price,Discount,RealPrice,PublisherID,PublisherName,PublishDate, "
        +" SmallImage,BigImage,Description,Contents,SaleSum,ClickCount "
        +" from ViewBookInfo where BookID=@id ";
    //创建命令并添加参数
    DbCommand command = db.GetSqlStringCommond(sql);
    db.AddInParameter(command, "@id", System.Data.DbType.Int32, bookid);
    //执行命令，得到数据读取器，读取数据
    using (DbDataReader reader = db.ExecuteReader(command))
    {
        if (!reader.Read())
            return null;                        //如果没有数据则返回空
       //循环读取各个字段，并赋值给 Book 类的实例
        Book book = new Book();
        book.id = bookid;
        book.title = reader["BookTitle"].ToString();
        book.author1 = Convert.ToString(reader["Author1"]);
        book.author2 = Convert.ToString(reader["Author2"]);
        book.author3 = Convert.ToString(reader["Author3"]);
        book.price = Convert.ToDouble(reader["Price"]);
        book.discount = Convert.ToDouble(reader["discount"]);
        book.realPrice = Convert.ToDouble(reader["RealPrice"]);
        book.publisherID = Convert.ToInt32(reader["PublisherID"]);
        book.publisherName = Convert.ToString(reader["PublisherName"]);
        book.smallImage = Convert.ToString(reader["SmallImage"]);
        book.bigImage = Convert.ToString(reader["BigImage"]);
        book.saleSum = Convert.ToInt32(reader["SaleSum"]);
        book.clickCount = Convert.ToInt32(reader["ClickCount"]);
        book.content = Convert.ToString(reader["Contents"]);
        book.description = Convert.ToString(reader["Description"]);
        book.publishDate = Convert.ToDateTime(reader["PublishDate"]);
        reader.Close();
        return book;
    }
}
```

BookDAL 类第二个功能是根据条件搜索图书信息。图书的查找条件比较复杂，可以根据书名、作者、出版社、出版日期、价格等任意组合条件查找。要实现这个功能，可以定义一个方法，搜索条件作为方法的参数，代码如下：

```
public static List<Book> search(string title, DateTime date1, DateTime date2,
string author,
string pulisher,  double price1, double price2);
```

由于查询条件太多，上面这个方法参数太多，方法签名太长，不方便调用。为了改善

这一情况，可以用一个类 SearchBookCriterion 来封装查询条件，则 search()方法只需要一个参数就可以。

```
// 搜索图书条件
public class SearchBookCriterion
{
    public string title=null;                          //标题
    public DateTime? date1=null, date2=null;           //出版日期
    public string author=null;                         //作者
    public string publisher=null;                      //出版社
    public double? price1=null, price2=null;           //价格区间
    //public int? category=null;
}
```

实现图书搜索时，需要根据查询条件动态构建对应的 Select 语句中的 Where 条件。实现思路为，如果搜索条件中题目不为空，则在 Where 子句中添加“BookTitle like @title”，如果搜索条件中题目为空，则不在 Where 子句中添加这个条件。其他查询条件也需要做类似处理。这个功能用 BookDAL 类的 buildWhereString()方法实现，代码如下：

```
/// <summary>
/// 根据查询条件构建 SQL 语句中的 where 子句
/// </summary>
/// <param name="criterion">查询条件</param>
/// <returns>构建好的 where 子句</returns>
private static string buildWhereString(SearchBookCriterion criterion)
{
    string sql = " ";
    string temp = null;
    //如果标题不为空，则在 select 的 where 子句中添加标题条件
    if (!string.IsNullOrEmpty(criterion.title))
    {
        temp = string.Format(" and (BookTitle like '%{0}%')  ", criterion.
        title);
        sql += temp;
    }
    //如果作者不为空，则在 select 的 where 子句中添加作者条件
    if (!string.IsNullOrEmpty(criterion.author))
    {
        temp = string.Format(" and (Author1 like '%{0}%' "
            + " or Author2 like  '%{0}%' "
            + " or Author2 like  '%{0}%' )  ", criterion.author);
        sql += temp;
    }
    //如果日期范围不为空，则在 select 的 where 子句中添加出版日期条件
    if (criterion.date1.HasValue && criterion.date2.HasValue)
    {
        temp = string.Format(" and (PublishDate between '{0}' and '{1}') ",
            criterion.date1.Value, criterion.date2.Value);
        sql += temp;
    }
    //如果价格范围不为空，则在 select 的 where 子句中添加价格条件
    if (criterion.price1.HasValue && criterion.price2.HasValue)
    {
        temp = string.Format(" and (Price between {0} and {1}) ",
            criterion.price1, criterion.price2);
        sql += temp;
    }
```

```
    //如果出版社不为空，则在 select 的 where 子句中添加出版社条件
    if (!string.IsNullOrEmpty(criterion.publisher))
    {
        temp = string.Format(" and (PublisherName like '%{0}%')  ",
        criterion.publisher);
        sql += temp;
    }
}
```

在实现图书搜索时，考虑到符合条件的图书会有很多种，通常需要把搜索结果分页返回。在 BookDAL 类的 search()方法中，需要传递相应参数 pageIndex 和 pageSize 以指定返回查询结果的第几页和每页记录数。

```
/// <summary>
/// 根据条件搜索图书，并分页返回结果
/// </summary>
/// <param name="criterion">搜索条件</param>
/// <param name="pageIndex">显示的当前页</param>
/// <param name="pageSize">页面大小</param>
/// <param name="count">符合条件的记录总数</param>
/// <returns>指定页的数据</returns>
public static DataTable search(SearchBookCriterion criterion,int
pageIndex,int pageSize,out int count)
{
    string where=" ";
    if(criterion!=null)
        where = buildWhereString(criterion);        //得到查询的 where 条件
    SqlHelper db = new SqlHelper();
    //用 select … + where … 构建完整的查询命令
    DbCommand command = db.GetSqlStringCommond("select count(*) from
    ViewBookInfo where 1=1 "+where);
    //得到满足查询条件的图书总数
    count = Convert.ToInt32(db.ExecuteScalar(command));
    //分页查询，查询指定页的图书列表
    string sql = "SELECT BookID,BookTitle,Price,Discount, RealPrice,
    Author1,Author2,Author3,"
       + " SmallImage,PublisherName,PublishDate,"
       + " Row Number() over (order by BookID desc) as rownum "
       + " FROM ViewBookInfo where (1=1) ";
    sql += where;
    string temp=null;
    sql = "WITH tempTable AS (" + sql + ") ";
    temp = string.Format(" select * from tempTable where rownum between {0}
    and {1} ",
        (pageIndex - 1) * pageSize + 1, pageIndex * pageSize);
    sql += temp;
    command = db.GetSqlStringCommond(sql);
    return db.ExecuteDataTable(command);
}
```

提示：上述代码中用到的 SQL Server 函数 Row_Number()在 SQL Server 2005 以上版本中才能够支持，如果读者所用的数据库为 SQL Server 2000，则需要使用其他分页方法，如利用 Not In 和 SELECT TOP 分页，具体代码可在网上查找。

上述代码中的 where (1=1)是动态构建 select 语句时一个常用小技巧。如果 SELECT 语句的 where 子句中列名称和列数量都不固定，那么在构建 Where 条件

时，第一个条件不需要添加 and，后面的条件都需要添加 and。这样在添加查询条件时，就需要不断判断是否第一个查询条件，从而决定是否需要 AND。而在 SELECT 后面添加 WHERE (1=1)很巧妙的解决了这个问题。

3．购物车类ShopCartItem和ShopCart

ShopCartItem 描述了购物车中的一种图书，这个类包括图书相关信息和所购买的图书数量。代码如下：

```
public class ShopCartItem
{
    public ShopCartItem() { }
    public ShopCartItem(Book theBook, int num)
    {
        book = theBook;
        number = num;
    }
    public Book book { get; set; }                          //所购买的图书
    public int bookID
    { get { return book.id; } }                             //图书编号
    public string bookTitle                                 //图书标题
    { get { return book.title; } }
    public double price                                     //价格
    { get { return book.price; } }
    public double discount                                  //折扣
    { get { return book.discount; } }
    public double realPrice                                 //实际购买价格
    { get { return book.realPrice; } }
    public string author1                                   //作者 1
    { get { return book.author1; } }
    public string author2                                   //作者 2
    { get { return book.author2; } }
    public string author3                                   //作者 3
    { get { return book.author3; } }
    public int number{get;set;}
    public double money                                     //金额=单价*数量
    {
        get { return realPrice * number; }
    }
}
```

ShopCart 类表示访问网站的一个用户的购物车。ShopCart 类是一个 ShopCartItem 的集合，可以向其中添加图书，还可以获取统计数据，如购物车图书总数和总金额等。ShopCart 类代码如下：

```
//购物车，保存在 Session 中
public class ShopCart
{
    //购物车中现有的图书列表
    private List<ShopCartItem> books=new  List<ShopCartItem>();
    public List<ShopCartItem> booksInCart
    {
        get
        {
            return books;
        }
```

```
    }
    /// <summary>
    /// 向购物车添加图书
    /// </summary>
    /// <param name="bookid">要添加的图书编号</param>
    /// <param name="number">要添加的图书数量</param>
    public void addBook(int bookid, int number)
    {
        //查找要添加的图书是否已经在购物车中存在，如果存在则只修改此图书的数量
        //如果不存在则需要在购物车中添加这个图书，并设置数量
        ShopCartItem item = books.Find(b => b.bookID == bookid);
        if (item != null)
        {
            item.number += number;
            if (item.number == 0)
                books.Remove(item);
            return;
        }
        Book book = BookDAL.getBookByID(bookid);       //从数据库获取图书信息
        if(book!=null)
            books.Add(new ShopCartItem(book, number));
    }
    //得到购物车中所有图书总金额
    public double totalMoney
    {
        get
        {
            double money = 0;
            foreach (var item in books)
            {
                money += item.money;
            }
            return money;
        }
    }
    //得到购物车中所有图书总数量
    public int bookNumSum
    {
        get
        {
            int sum = 0;
            foreach (var item in books)
            {
                sum += item.number;
            }
            return sum;
        }
    }
    // 得到购物车中图书种类数
    public int bookCount
    {
        get { return books.Count; }
    }
}
```

4. 常量容器类Constants

在网上书店项目的多个页面中都需要访问一些常量，例如，需要访问 Session 的一些

键值，以取得 Session 中存储的相应对象。把这些常量封装到一个类中集中存储，既方便了代码的编写和维护，又能够避免编程时 Session 键值拼写出错。

```
internal static class Constants
{
    internal const string SessionAdmin = "AdminLogin";
    internal const string SessionMember = "MebmerLogin";
    internal const string SessionShopCart = "ShoppingCart";
}
```

4.2.3　网书列表用户控件

在网上书店的多个页面中都需要显示图书列表，为了实现复用，可以把图书列表做成一个用户控件 BookListControl，在其中定义好图书显示布局，实现相应的事件处理程序。当在页面中需要显示图书列表时，只需要使用此控件即可。

BookListControl 用户控件使用 Repeater 控件显示图书列表。Repeater 控件的 ItemTemplate 包含左右两大部分，左侧为图书封面图片，右侧为图书详细文字描述，还有放到购物车的按钮购买数量。图书封面为超链接，用户可以通过单击图书封面转到图书详情页面。BookListControl 控件显示效果如图 4.3 所示。



图 4.3　图书列表用户控件 BookListControl

BookListControl 控件代码如下：

```
<asp:Repeater ID="Repeater1" runat="server"  OnItemCommand="Repeater1_
ItemCommand">
<HeaderTemplate>
<table>
</HeaderTemplate>
<ItemTemplate>
<tr>
<td>
<a href="BookDetail.aspx?id=<%#Eval("BookID")%>">
<img src="BookImages/<%#Eval("SmallImage") %>" alt="<%#Eval("BookTitle")
%>"
    style="width:100px;" /> </a>
</td>
<td>
<h3> <%#Eval("BookTitle") %> </h3>
作者：<%#Convert.ToString(Eval("Author1")) + Convert.ToString (Eval
```

```
("Author2")) +
   Convert.ToString(Eval("Author3"))%>|
出版社：<%#Eval("PublisherName") %>|  出版日期：<%#Eval("PublishDate") %><br
/><br />
定价：<span style="color:Blue;"> <%#Eval("Price","{0:C}") %></span>  |
会员价：<span style="color:red;"><%#Eval("RealPrice","{0:C}")%></span>|
折扣：<%#Eval("Discount")%><br /><br />
购买数量：<asp:TextBox ID="buyNum" runat="server" Text="1"></asp:TextBox>
<asp:ImageButton runat="server" ID="buyButton" ImageUrl="~/images/buy.
png" Height="25"
CommandName="buy" CommandArgument='<%#Eval("BookID") %>' ImageAlign="top" />
<span style="margin-left:50px;">
<a href="BookDetail.aspx?id=<%#Eval("BookID")%>">查看详情</a>
</span>
</td>
</tr>
</ItemTemplate>
<FooterTemplate>
</table>
</FooterTemplate>
</asp:Repeater>
```

图书列表控件 BookListControl 将在多个页面中使用，需要显示各种图书数据，所以 BookListControl 控件本身不能确定其数据源，而是需要定义一个接口，允许外部页面为其绑定数据源。在 BookListControl 控件中添加一个 bindData 方法实现这个功能。

```
public void bindData(DataTable table)
{
   Repeater1.DataSource = table;
   Repeater1.DataBind();
}
```

当用户在图书列表中单击“加入购物车”按钮时，会触发 Repeater 控件的 ItemCommand 事件，在此事件中编写代码，获取用户输入的数量，将用户所选的图书添加到 Session 中的购物车。代码如下。

```
protected void Repeater1 ItemCommand(object source, RepeaterCommand-
EventArgs e)
{
   if (e.CommandName == "buy")
   {
      TextBox text=e.Item.FindControl("buyNum") as TextBox;
                                            //找到数量 TextBox 控件
      int n;
      if (!int.TryParse(text.Text, out n))
         n = 1;
      int id = int.Parse(e.CommandArgument.ToString());
      //从 Session 中获取购物车，并将图书添加到其中
      if (Session[Constants.SessionShopCart] == null)
         Session[Constants.SessionShopCart] = new ShopCart();
      ShopCart cart = Session[Constants.SessionShopCart] as ShopCart;
      cart.addBook(id, n);
   }
}
```

4.2.4　网站首页

当用户登录网上书店首页时，应该向用户展示某些图书信息，例如最新推出的图书或者最近销售量最大的图书。同时，首页上应该允许用户方便地找到自己所需要的图书，应该向用户提供按照关键字搜索图书的功能。其中，搜索图书按钮已经包含在母版页中，首页中需要做的主要工作是显示推荐的图书列表，本例将显示最新添加的图书。

（1）在数据库中添加视图 ViewBookInfo 以获取图书详细信息。

```
CREATE VIEW ViewBookInfo
AS
SELECT dbo.Book.BookID, dbo.Book.BookTitle, dbo.Book.Author1, dbo.Book.
Author2,
     dbo.Book.Author3, dbo.Book.PublisherID, dbo.Book.Price, dbo.Book.
     Discount,
     dbo.Book.PublishDate, dbo.Book.Description, dbo.Book.Contents,
     dbo.Book.SaleSum,
     dbo.Book.ClickCount, dbo.Book.SmallImage, dbo.Book.BigImage,
     dbo.Book.RealPrice,
     dbo.Publisher.PublisherName
FROM dbo.Book INNER JOIN
     dbo.Publisher ON dbo.Book.PublisherID = dbo.Publisher.PublisherID
```

（2）在数据库中添加一个存储过程以分页读取最新添加的图书。

```
CREATE PROCEDURE dbo.GetNewBook
    @pageIndex int=1,                                   /*要返回的页码*/
    @pageSize int=10,                                   /*页面大小*/
    @bookCount int output                               /*输出参数，记录总数*/
AS
    declare @min int , @max int;
    set @min=(@pageIndex-1)*@pageSize+1;
    set @max=@pageIndex*@pageSize;
    select @bookCount=count(*) from Book;
    WITH NewBooks AS
    (SELECT
BookID,BookTitle,Price,Discount,RealPrice,Author1,Author2,Author3,
    SmallImage,PublisherName,PublishDate,
    Row Number() over (order by BookID desc) as rownum
    FROM ViewBookInfo)
    select
BookID,BookTitle,Price,Discount,RealPrice,Author1,Author2,Author3,
    SmallImage,PublisherName,PublishDate
     from NewBooks where rownum between @min and @max ;
    RETURN
```

（3）使用 BookShop.master 作为母版页添加一个新页面 Default.aspx，并在页面上放置一个图书列表用户控件 BookListControl 和一个第三方分页控件 AspNetPager，页面代码如下：

```
<%@ Register Assembly="AspNetPager" Namespace="Wuqi.Webdiyer" TagPrefix=
"webdiyer" %>
<%@ Register src="BookListControl.ascx" tagname="BookListControl"
tagprefix="uc1" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
```

```
    <uc1:BookListControl ID="bookList" runat="server" />
    <webdiyer:AspNetPager ID="AspNetPager1" runat="server">
    </webdiyer:AspNetPager>
</asp:Content>
```

（4）在 Default.aspx 页面的 Page_Load 事件中编写代码显示第一页最新图书信息。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        //页面首次加载时，设置分页控件默认大小为 10，当前页码为 1，并绑定第 1 页数据
        AspNetPager1.PageSize = 10;
        AspNetPager1.CurrentPageIndex = 1;
        bindData();
    }
}
private void bindData()
{
    SqlHelper db = new SqlHelper();
    //调用 GetNewBook 存储过程
    DbCommand command= db.GetStoredProcCommond("GetNewBook");
    //为存储过程添加参数：2 个输入参数（页号和页面大小），1 个输出参数（记录总数）
    db.AddInParameter(command, "@pageIndex", DbType.Int32, AspNetPager1.
    CurrentPageIndex);
    db.AddInParameter(command, "@pageSize", DbType.Int32, AspNetPager1.
    PageSize);
    db.AddOutParameter(command, "@bookCount", DbType.Int32, 4);
    DataTable table= db.ExecuteDataTable(command);
    bookList.bindData(table);
    //设置 AspNetPager 分页控件的记录总数
    AspNetPager1.RecordCount = Convert.ToInt32(command. Parameters
    ["@bookCount"].Value);
}
```

（5）在 AspNetPager 控件的 PageChanged 事件中重新绑定新页面的数据。

```
protected void AspNetPager1 PageChanged(object sender, EventArgs e)
{
    bindData();
}
```

（6）运行 Default.aspx 页面，运行效果如图 4.4 所示。



图 4.4　攀登者网上书店首页

4.2.5　购物车

攀登者网上书店可以随时从页面顶部链接进入购物车页面，查看和修改购物车中的图书商品。由于一个用户对应于一个购物车，而且需要随时访问购物车，因此把购物车存放在 Session 中是一个很好的解决方案。当进入购物车页面时，需要从 Session 而不是数据库中读取购物车数据。

（1）使用 BookShop.master 作为母版页添加一个新页面 ShopCartPage.aspx。

（2）在 ShopCartPage 页面上放置一个 GridView 以显示购物车数据。GridView 的最后两列分别是模板列和超链接列，模板列中用一个 TextBox 控件显示了图书数量，用户可以编辑此数量。用户如果单击超链接列会转移到图书详情页面。

```
<asp:GridView ID="GridView1" runat="server" DataKeyNames="bookID"
    AutoGenerateColumns="false" ShowFooter="true"
    onrowupdating="GridView1_RowUpdating" onrowcreated="GridView1_
    RowCreated" >
     <%--设置 GridView 的空模板，当没有数据时就显示此模板的内容--%>
    <EmptyDataTemplate>
    <div class="dashedborder" style="color:navy; padding:20px 30px;
    font-size:20px;">
    你的购物车中没有商品。
    </div>
    </EmptyDataTemplate>
<%--以下为 GridView 的各个列--%>
<Columns>
<asp:BoundField DataField="bookID"  HeaderText="图书编号"/>
<asp:BoundField DataField="bookTitle"  HeaderText="图书题目" ItemStyle-
Width="260"/>
<asp:BoundField DataField="price" DataFormatString="{0:C}"  HeaderText="
定价"  ItemStyle-Width="80"/>
<asp:BoundField DataField="discount"  HeaderText="会员折扣" ItemStyle-
Width="80"/>
<asp:BoundField DataField="realPrice" DataFormatString="{0:C}"
HeaderText="会员价格"  ItemStyle-Width="80"/>
<asp:BoundField DataField="number"   HeaderText="购买数量"  ItemStyle-
Width="60"/>
<asp:BoundField DataField="money" DataFormatString="{0:C}"  HeaderText="
金额"
    ItemStyle-Width="80"/>
<%--   <asp:TemplateField HeaderText="作者"  ItemStyle-Width="200">
<ItemTemplate>
<%#Convert.ToString(Eval("author1")) + Convert.ToString(Eval("author2"))
+
Convert.ToString(Eval("author3"))%>
</ItemTemplate>
</asp:TemplateField>  --%>
<asp:TemplateField HeaderText="修改数量" >
<ItemTemplate>
<asp:TextBox runat="server" ID="buyNum" Text='<%#Bind("number")%>'
Width="80" />
<asp:Button
ID="Button1" runat="server" Text="修改" CommandName="update"
    CommandArgument='<%#Bind("bookID")%>' />
```

```
</ItemTemplate>
</asp:TemplateField>
<asp:HyperLinkField Text="查看" HeaderText="详情"
       DataNavigateUrlFields="BookID"
       DataNavigateUrlFormatString="BookDetail.aspx?id={0}" />
</Columns>
</asp:GridView>
```

（3）在 Page_Load 事件中，获取 Session 中保存的购物数据，并显示在 GridView 控件中。

```
protected void Page_Load(object sender, EventArgs e)
{
    if(!IsPostBack)
       bindData();
}
private void bindData()
{
    //从 Session 中获取购物车，并将其中的图书信息显示在 GridView 中
    if (Session[Constants.SessionShopCart] == null)
       return;
    ShopCart cart = Session[Constants.SessionShopCart] as ShopCart;
    GridView1.DataSource = (Session[Constants.SessionShopCart] as
    ShopCart).booksInCart;
    GridView1.DataBind();
}
```

（4）在 GridView 的 RowCreated 事件中，生成购物车汇总数据并显示在 GridView 的 Footer 中。

```
protected void GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    //汇总数据在页脚中，如果当前行不是页脚，则不进行处理，方法返回
    if (e.Row.RowType != DataControlRowType.Footer)
       return;
    //清除页脚原有单元格，创建一个跨越所有列的大单元格
    int n=e.Row.Cells.Count;
    e.Row.Cells.Clear();
    TableCell cell = new TableCell();
    cell.ColumnSpan = n;
    ShopCart cart = Session[Constants.SessionShopCart] as ShopCart;
    //在单元格中显示汇总信息
    cell.Text = string.Format("购物车中总计{0}种{1}本图书，总金额{2:C}",
       cart.bookCount,cart.bookNumSum,cart.totalMoney);
    e.Row.Cells.Add(cell);
}
```

（5）当用户修改购物车数量时，会触发 GridView 的 RowUpdating 事件，在此事件中修改 Session 中保存的购物车图书数量。

```
protected void GridView1 RowUpdating(object sender, GridViewUpdate-
EventArgs e)
{
    string id = e.Keys[0].ToString();
    //得到购买数量 TextBox 控件
    TextBox control = GridView1.Rows[e.RowIndex].FindControl("buyNum") as
    TextBox;
    ShopCart cart = Session[Constants.SessionShopCart] as ShopCart;
    int num=int.Parse(control.Text);
```

```
    var book=cart.booksInCart.Find(b => b.bookID == int.Parse(id));
                                                          //查找购物车中此图书
    if (num > 0)
        book.number = num;
    else
        cart.booksInCart.Remove(book);
    bindData();                                           //重新绑定数据
}
```

（6）购物车页面运行界面如图 4.5 所示。



图 4.5　购物车页面

4.2.6　简单搜索

攀登者网上书店可以方便地根据书名或者作者搜索图书。用户在页面顶部的搜索框中输入关键字，单击“搜索”按钮，即可查找书名或者作者名字中包含此关键字的图书。搜索框和搜索按钮是在母版页 BookShop.master 中定义的，当单击“搜索”按钮时，会调用母版页中以下 JavaScript 代码，检查用户是否输入了关键字，并将搜索关键字作为 QueryString 参数传递给 SimpleSearch.aspx 页面，在 SimpleSearch.aspx 页面中实现图书搜索和显示功能。

```
<script type="text/javascript" language="javascript">
    function simpleSearch() {
        //获取用户输入的关键字，验证是否为空，如果为空则提示
        var key = document.getElementById("keyWord").value;
        if (!key) {
            alert('请输入关键字再搜索');
            return;
        }
        //跳转到简单搜索页面，并将搜索关键字作为参数传递
        window.location = "SimpleSearch.aspx?key=" + key;
    }
</script>
```

（1）在数据库中添加一个存储过程 SimpleSearch，实现简单搜索功能。

```
CREATE PROCEDURE dbo.SimpleSearchBook
    /* 根据书名或者作者搜索图书 */
    @title_author nvarchar(50),                             /*书名或作者*/
    @pageIndex int=1,                                       /*页码*/
    @pageSize int=10,                                       /*页面大小*/
    @bookCount int output                                   /*查找到的总数量*/
AS
    //计算指定页面的最小行号和最大行号
    declare @min int , @max int;
    set @min=(@pageIndex-1)*@pageSize+1;
    set @max=@pageIndex*@pageSize;
    //执行查询
    set @title_author='%'+@title_author+'%'
    select @bookCount=count(*) from Book
    where (BookTitle like @title_author)
    or (author1 like @title_author)
    or (author2 like @title_author)
    or (author3 like @title_author);
    WITH NewBooks AS
    (SELECT BookID,BookTitle,Price,Discount, RealPrice,Author1, Author2,
    Author3,
    SmallImage,PublisherName,PublishDate,
    Row_Number() over (order by BookID desc) as rownum
    FROM ViewBookInfo where BookTitle like '%'+@title_author+'%' )
    select BookID,BookTitle,Price,Discount,RealPrice, Author1, Author2,
    Author3,
    SmallImage,PublisherName,PublishDate
    from NewBooks where rownum between @min and @max;
```

（2）使用 BookShop.master 作为母版页添加一个新页面 SimpleSearch.aspx，并在页面上放置一个图书列表用户控件 BookListControl 和一个第三方分页控件 AspNetPager，页面代码如下：

```
<%@ Register Assembly="AspNetPager" Namespace="Wuqi.Webdiyer" TagPrefix=
"webdiyer" %>
<%@ Register src="BookListControl.ascx" tagname="BookListControl"
tagprefix="uc1" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
   <uc1:BookListControl ID="bookList" runat="server" />
   <webdiyer:AspNetPager ID="AspNetPager1" runat="server"
      onpagechanged="AspNetPager1_PageChanged">
   </webdiyer:AspNetPager>
</asp:Content>
```

（3）在 Page_Load 事件中，获取 QuerySting 中的查询关键字，调用存储过程搜索图书，并显示结果。

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
   {
      string key = Request.QueryString["key"];          //得到查询关键字
      if (string.IsNullOrEmpty(key)) return;
      AspNetPager1.CurrentPageIndex = 1;
      bindData();                                       //执行查询
```

```
    }
}
private void bindData()
{
    string key = Request.QueryString["key"];        //得到查询关键字
    SqlHelper db = new SqlHelper();
    //调用 SimpleSearchBook 存储过程查询图书
    var command=db.GetStoredProcCommond("SimpleSearchBook");
    db.AddInParameter(command, "@title_author", System.Data.DbType.
    String, key);
    db.AddInParameter(command, "@pageIndex", System.Data.DbType.Int32,
    AspNetPager1.CurrentPageIndex);
    db.AddInParameter(command, "@pageSize", System.Data.DbType.Int32,
    AspNetPager1.PageSize);
    db.AddOutParameter(command, "@bookCount", System.Data.DbType.Int32,4);
    var table=db.ExecuteDataTable(command);
    //将查询到的结果绑定到 GridView，并设置分页控件的总记录数
    bookList.bindData(table);
    int count= Convert.ToInt32(command.Parameters["@bookCount"].Value);
    AspNetPager1.RecordCount = count;
    //若查询到的总记录数为 0，则提示没有符合条件的数据
    if (AspNetPager1.RecordCount == 0)
    {
        ClientScript.RegisterStartupScript(this.GetType(), "nobooks",
            "<script>没有符合条件的图书。</script>");
    }
}
```

（4）在 AspNetPager 控件的 PageChanged 事件中显示新页面数据。

```
protected void AspNetPager1_PageChanged(object sender, EventArgs e)
{
    bindData();
}
```

4.2.7　高级搜索

在高级搜索页面中，用户可以根据书名、作者、出版社、出版日期、价格区间，以及这些条件的任意组合搜索图书。

（1）使用 BookShop.master 作为母版页添加一个新页面 AdvanceSearch.aspx。

（2）在 AdvanceSearch.aspx 页面上放置用于输入各个搜索条件的 TextBox 控件。

```
<div class="dashedborder">
标题：<asp:TextBox ID="bookTitle" runat="server"/>
 作者：<asp:TextBox ID="author" runat="server"/>
 出版社：<asp:TextBox ID="publisher" runat="server"/> <br />
价格范围：<asp:TextBox ID="price1" runat="server"></asp:TextBox>至<asp:
TextBox ID="price2" runat="server"></asp:TextBox>
 
出版日期：<asp:TextBox ID="date1" runat="server"></asp:TextBox>至<asp:
TextBox ID="date2" runat="server"></asp:TextBox>
     <asp:Button ID="Button1" runat="server" Text="搜索" Width="80"
        onclick="Button1_Click" />
</div>
```

（3）在页面上放置一个图书列表用户控件 BookListControl 和一个第三方分页控件 AspNetPager。

```
<%@ Register Assembly="AspNetPager" Namespace="Wuqi.Webdiyer" TagPrefix=
"webdiyer" %>
<%@ Register src="BookListControl.ascx" tagname="BookListControl"
tagprefix="uc1" %>
   <uc1:BookListControl ID="bookList" runat="server" />
   <webdiyer:AspNetPager ID="AspNetPager1" runat="server"
      onpagechanged="AspNetPager1_PageChanged">
   </webdiyer:AspNetPager>
```

（4）在“搜索”按钮的 Click 事件中，根据用户输入的搜索条件查找并显示符合条件的第一页数据。要注意把查询条件保存到 Session 中，从而可以在用户翻页时重新获取并继续使用此查询条件。

```
private void bindData()
{
   //从 Session 中获取查询条件
   SearchBookCriterion criterion = Session["SearchCriterion"] as
   SearchBookCriterion;
   int index = AspNetPager1.CurrentPageIndex;
   int size = AspNetPager1.PageSize;
   int count;
  //调用数据访问层代码执行搜索操作，并将结果绑定到控件
   DataTable table = BookDAL.search(criterion, index, size, out count);
   BookListControl1.bindData(table);
   AspNetPager1.RecordCount = count;
}
protected void Button1_Click(object sender, EventArgs e)
{
   //根据用户输入的查询条件构建一个 SearchBookCriterion 实例
   SearchBookCriterion criterion = new SearchBookCriterion();
   if (!string.IsNullOrEmpty(bookTitle.Text))
      criterion.title = bookTitle.Text;
   if (!(string.IsNullOrEmpty(price1.Text) || string.IsNullOr-
   Empty(price2.Text)))
   {
      criterion.price1 = double.Parse(price1.Text);
      criterion.price2 = double.Parse(price2.Text);
   }
   if (!string.IsNullOrEmpty(author.Text))
      criterion.author = author.Text;
   if (!string.IsNullOrEmpty(publisher.Text))
      criterion.publisher = publisher.Text;
   if (!(string.IsNullOrEmpty(date1.Text) || string.IsNullOrEmpty
   (date2.Text)))
   {
      criterion.date1 = DateTime.Parse(date1.Text);
      criterion.date2 = DateTime.Parse(date2.Text).AddDays
      AddSeconds(-1);
   }
   Session["SearchCriterion"] = criterion;
   //将当前页码设置为 1，根据新条件重新绑定数据
   AspNetPager1.CurrentPageIndex = 1;
   bindData();
}
```

（5）在 AspNetPager 的 PageChanged 事件中显示新的一页数据。

```
protected void AspNetPager1_PageChanged(object sender, EventArgs e)
{
    bindData();
}
```

（6）高级搜索页面 AdvanceSearch.aspx 运行界面，如图 4.6 所示。



图 4.6　高级搜索页面

4.3　网上书店后台功能实现

管理员登录网上书店后台，可以对图书信息进行全面管理，如添加、删除图书，修改图书文字资料，上传图书封面图片等。为了维护网站数据安全，只能经过登录验证的管理员才能访问后台管理页面，在实现后台页面时需要考虑身份验证的问题。为了便于进行身份验证，所有后台管理页面都在网站根目录下的 admin 文件夹中。

4.3.1　用户身份验证模块

实现用户身份验证最简单的方法就是用户登录后把用户信息保存在 Session 中，当用户访问每个页面时，都检查 Session 以确认用户身份，如果身份不符则拒绝访问。由于后台管理页面很多，如果在每个页面的 Page_Load 事件中都编写代码检测用户身份，工作量大而且代码重复。ASP.NET 提供了一种 HttpModule 的机制可以很好地解决这个问题。

HttpModule 是 ASP.NET 的一种 HTTP 处理机制，当把 HttpModule 注册到网站或者 Web 应用中以后，每当接到一个新的 HTTP 请求时，都会执行 HttpModule 中的相应代码。利用这种机制，可以把所有页面公用的代码写到 HttpModule 中。

攀登者网上书店项目正是使用了 HttpModule 实现用户身份验证。具体思路为：每当有新的 HTTP 请求到达时，获取所请求 URL，并分析此 URL 所对应页面是否在网上书店根目录的 Admin 文件夹中。如果在 Admin 文件夹中，则说明用户请求访问后台管理页面，

需要验证用户身份；如果不在 Admin 文件夹中，则说明用户访问的是普通页面，不需要身份验证。另外，Admin 文件夹中有两个特殊页面 AdminLogin 和 Logout 不需要身份验证。代码如下：

```
public class CheckAdminModule : IHttpModule
{
    #region IHttpModule 成员
    public void Dispose()
    {    }
    public void Init(HttpApplication context)
    {
        context.AcquireRequestState += new EventHandler(OnRequest);
    }
    #endregion
    public void OnRequest(Object source, EventArgs e)
    {
        Uri url = HttpContext.Current.Request.Url;      //得到所请求的 URL
        //请求 Admin 目录下的文件时，需要进行身份验证，只有管理员才能访问
        if (url.AbsolutePath.ToLower().StartsWith("/admin"))
        {
            //adminlogin.aspx 和 logout.aspx 不需要身份验证
            if (url.AbsolutePath.ToLower().EndsWith("adminlogin.aspx"))
                return;
            if (url.AbsolutePath.ToLower().EndsWith("logout.aspx"))
                return;
            //如果 Session 中没有保存管理员信息，则转到管理员登录页面
            if (HttpContext.Current.Session[Constants.SessionAdmin] ==
            null)
            {
                HttpContext.Current.Response.Redirect("AdminLogin.
                aspx?url="
                    +HttpContext.Current.Server.HtmlEncode(url.
                    PathAndQuery));
            }
        }
    }
}
```

HttpModule 需要在 web.config 文件中注册后才能起作用。在 web.config 文件中添加以下代码注册 CheckAdminUser。

```
<httpModules>
    <add name="CheckAdminModule" type="BookShop.Admin.CheckAdminModule"/>
</httpModules>
```

4.3.2　管理员登录和修改密码

管理员登录时，输入用户名和密码，登录成功后，将登录信息保存到 Session 中。管理登录页面 AdminLogin 代码如下：

```
//AdminLogin.aspx
<form id="form1" runat="server">
<div>
<img src="../images/adminlogin.jpg" alt="管理员登录" style="width:500px;"
/><br />
用户名：<asp:TextBox ID="username" runat="server" />
```

```
<%--验证控件，必须输入用户名--%>
<asp:RequiredFieldValidator ID="RequiredFieldValidator1" Display="None"
    runat="server" ErrorMessage="必须输入用户。" ControlToValidate=
    "username">
</asp:RequiredFieldValidator>
密码：<asp:TextBox ID="password" runat="server" TextMode="Password" />
<%--验证控件，必须输入用户名--%>
<asp:RequiredFieldValidator
    ID="RequiredFieldValidator2" runat="server" ErrorMessage="必须输入密码。"
     ControlToValidate="password" Display="None"></asp:
     RequiredFieldValidator>
    <asp:Button ID="Button1" runat="server" Text="登录" onclick=
    "Button1_Click" />
    <asp:ValidationSummary ID="ValidationSummary1" runat="server"
    ForeColor="Red" />
</div>
</form>
// AdminLogin.aspx.cs
public partial class AdminLogin : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {   }
    protected void Button1_Click(object sender, EventArgs e)
    {
        SqlHelper db = new SqlHelper();
        //从数据库验证用户名和密码是否正确
        DbCommand command = db.GetSqlStringCommond(
            "select count(*) from AdminUser where AdminID=@id and Password=
            @pass");
        db.AddInParameter(command, "@id", DbType.String, username.Text);
        db.AddInParameter(command, "@pass", DbType.String, password.Text);
        int n = Convert.ToInt32(db.ExecuteScalar(command));
       //如果用户不存在则在页面上弹出提示消息
        if (n == 0)
        {
            Session[Constants.SessionAdmin] = null;
            Page.ClientScript.RegisterStartupScript(this.GetType(),
            "errorlogin",
                "<script>alert('你输入的用户名或密码不正确。');</script>");
            return;
        }
        //登录成功后，将登录信息保存到Session中，并跳转到所请求的页面
        Session[Constants.SessionAdmin] = username.Text;
        if (Request.QueryString["url"] != null)
            Response.Redirect(Server.HtmlDecode(Request. QueryString
             ["url"]));
        else
            Response.Redirect("BookAdmin.aspx");
    }
}
```

在修改密码页面 AdminChangePass.aspx 中，用户输入原密码和两次新密码进行密码修改。在页面上除 3 个密码框外，还有 3 个 RequiredFieldValidator 验证控件保证输入密码不为空，一个 CompareValidator 验证控件保存两次新密码一致。页面代码如下：

```
//AdminChangePass.aspx
<form id="form1" runat="server">
<div>
```

```
<h3>修改密码</h3>
原密码：<asp:TextBox ID="oldPass" runat="server" TextMode="Password">
</asp:TextBox>
    <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat=
    "server"
        ErrorMessage="原密码不能为空" ControlToValidate="oldPass">
</asp:RequiredFieldValidator>
    <br />
新密码：<asp:TextBox ID="newPass1" runat="server" TextMode="Password"
></asp:TextBox>
    <asp:RequiredFieldValidator ID="RequiredFieldValidator2"
    runat="server"
        ErrorMessage="新密码不能为空" ControlToValidate="newPass1">
    </asp:RequiredFieldValidator>
    <br />
新密码：<asp:TextBox ID="newPass2" runat="server" TextMode="Password"><
/asp:TextBox>
    <asp:RequiredFieldValidator ID="RequiredFieldValidator3" runat=
    "server"
        ErrorMessage="新密码不能为空"  ControlToValidate="newPass2">
    </asp:RequiredFieldValidator>
    <asp:CompareValidator ID="CompareValidator1" runat="server"
        ErrorMessage="两次输入新密码不一致"  ControlToValidate="newPass2"
        ControlToCompare="newPass1"></asp:CompareValidator>
    <br />
    <asp:Button ID="Button1" runat="server" Text="修改密码" onclick=
    "Button1_Click" />
    <input type="button" value="关闭窗口" onclick="window.close();" />
</div>
</form>
//AdminChangePass.aspx.cs
public partial class AdminChangePass : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {  }
    protected void Button1 Click(object sender, EventArgs e)
    {
        //获取 Session 中保存的管理员登录信息
        string user = Convert.ToString(Session[Constants.SessionAdmin]);
        //从数据库查询原密码
        string sql = "select Password from AdminUser where AdminID=@admin";
        SqlHelper db = new SqlHelper();
        System.Data.Common.DbCommand command = db.GetSqlStringCommond(sql);
        db.AddInParameter(command,"@admin",System.Data.DbType.String,
        user);
        string old = db.ExecuteScalar(command).ToString();
       //比较原密码输入是否正确，如果不正确则提示，不设置新密码
        if (old != oldPass.Text)
        {
            ClientScript.RegisterStartupScript(this.GetType(), "fail",
            "<script>alert('输入的原始密码不正确！');</script>");
            return;
        }
        //原密码输入正确，则修改新密码
        sql = "update AdminUser set Password=@pass where AdminID=@admin";
        command = db.GetSqlStringCommond(sql);
        db.AddInParameter(command, "@pass", System.Data.DbType.String,
        newPass1.Text);
        db.AddInParameter(command, "@admin", System.Data.DbType.String,
```

```
            user);
            db.ExecuteNonQuery(command);
            ClientScript.RegisterStartupScript(this.GetType(), "suceess",
                "<script>alert('密码修改成功! ');</script>");
        }
}
```

4.3.3　后台管理母版页

网站后台和前台使用不同的页面布局和导航菜单，所以后台管理页面不能使用前台的母版页，而是要单独定义母版页。后台母版页结构比较简单，主要包括功能导航菜单。当用户在导航菜单中选择“修改密码”时，会调用 JavaScript 代码弹出一个模式对话框。后台母版页 Admin.master 文件代码如下：

```
<head runat="server">
    <title>攀登者网上书店</title>
    <script type="text/javascript" language="javascript">
        function changePassDialog() {
window.showModalDialog("AdminChangePass.aspx","","dialogWidth:500px;dia
logHeight:300px;");
        }
    </script>
</head>
<body>
    <form id="form1" runat="server">
    <div id="page">
    <div id="header" style="background-color:#e0f0ff; border:1px solid
    green; height:40px; ">
    <div style="float:left;">
      <img src="../images/logo.jpeg" alt="攀登者网上书店" style="height:30px;
      vertical-align:bottom;" />
      <span style="font-size:20px; font-weight:bold;">欢迎使用网上书店后台管理
      </span>
      </div>
      <div style="float:right; margin-top:10px;">
      <a href="BookAdmin.aspx">图书管理</a> |
      <a href="CategoryAdmin.aspx">类别管理</a> |
      <a href="#">会员管理</a> |
      <a href="#">订单管理</a> |
       <a href="#">销售统计</a>|
      <a href="javascript:changePassDialog();">修改密码</a> |
      <a href="../Default.aspx">返回首页</a> |
      <a href="Logout.aspx">退出登录</a>
      </div>
     </div>
     <div style="clear:both;">
        <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">

        </asp:ContentPlaceHolder>
        </div>
    </div>
    <div id="footer" />
    </form>
</body>
</html>
```

后台管理母版页界面如图 4.7 所示。

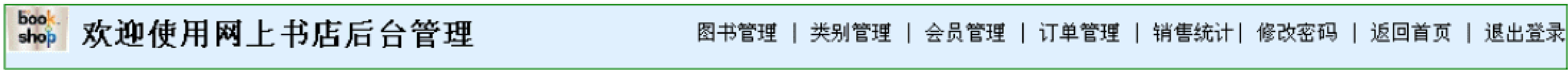


图 4.7　后台管理母版页

4.3.4　图书类别管理

在图书类别管理页面 CategoryAdmin.aspx 中，管理员可以添加、删除、修改图书类别。由于图书类别数据量少，而且使用这个页面的用户少（只有管理员才用），在这个页面中可以使用 GridView 和 SqlDataSource 自带的数据编辑、删除和分页功能，不会影响应用程序性能。

（1）以 Admin.master 作为母版页添加一个新页面 CategoryAdmin.aspx。

（2）在页面上放置一个 GridView 和 SqlDataSource。配置 SqlDataSource 使其读取数据库中 Category 表的数据，并生成 INSERT、DELETE、UPDATE 语句。设置 GridView 控件的数据源为 SqlDataSource，并设置各列标题，启用编辑和删除。

```
<br /><b>现有图书类别</b><br />
<asp:GridView ID="grid" runat="server" AutoGenerateColumns="False"
   DataKeyNames="CategoryID" DataSourceID="sqlData1" AllowPaging="True">
   <Columns>
      <asp:CommandField ShowDeleteButton="True" ShowEditButton="True"
      EditText="编辑" UpdateText="保存" CancelText="取消" DeleteText="删除
      " HeaderText="操作"  />
      <asp:BoundField DataField="CategoryID" HeaderText="类别编号"
      ReadOnly="True"
         SortExpression="CategoryID" ItemStyle-Width="100"  ItemStyle-
         HorizontalAlign="Center"/>
      <asp:BoundField DataField="CategoryName" HeaderText="类别名称"
         SortExpression="CategoryName" ItemStyle-Width="300" ItemStyle-
         HorizontalAlign="Center" />
   </Columns>
</asp:GridView><br /><br />
<asp:SqlDataSource ID="sqlData1" runat="server" ConnectionString="<%$
ConnectionStrings:database%>"
   DeleteCommand="DELETE FROM [Category] WHERE [CategoryID] = @CategoryID"
   InsertCommand="INSERT INTO [Category] ([CategoryName]) VALUES
   (@CategoryName)"
   ProviderName="<%$ ConnectionStrings:database.ProviderName %>"
   SelectCommand="SELECT [CategoryID], [CategoryName] FROM [Category]"
   UpdateCommand="UPDATE [Category] SET [CategoryName] = @CategoryName
   WHERE [CategoryID] = @CategoryID">
   <DeleteParameters>
      <asp:Parameter Name="CategoryID" Type="Int32" />
   </DeleteParameters>
   <InsertParameters>
      <asp:Parameter Name="CategoryName" Type="String" />
   </InsertParameters>
   <UpdateParameters>
      <asp:Parameter Name="CategoryName" Type="String" />
      <asp:Parameter Name="CategoryID" Type="Int32" />
   </UpdateParameters>
</asp:SqlDataSource>
```

（3）在页面上添加一个 DetailsView 控件以添加新的图书类别。设置 DetailsView 的数据源为 SqlDataSource，设置默认模式为插入模式。

```
<b>添加新类别</b><br />
<asp:DetailsView ID="DetailsView1" runat="server" DataSourceID="sqlData1"
   DefaultMode="Insert" AutoGenerateRows="False" >
<Fields>
   <asp:BoundField DataField="CategoryName" HeaderText="类别名称" />
   <asp:CommandField ShowInsertButton="True" InsertText="添加"
   ShowCancelButton="false" />
</Fields>
</asp:DetailsView>
```

（4）CategoryAdmin.aspx 页面运行界面如图 4.8 所示。

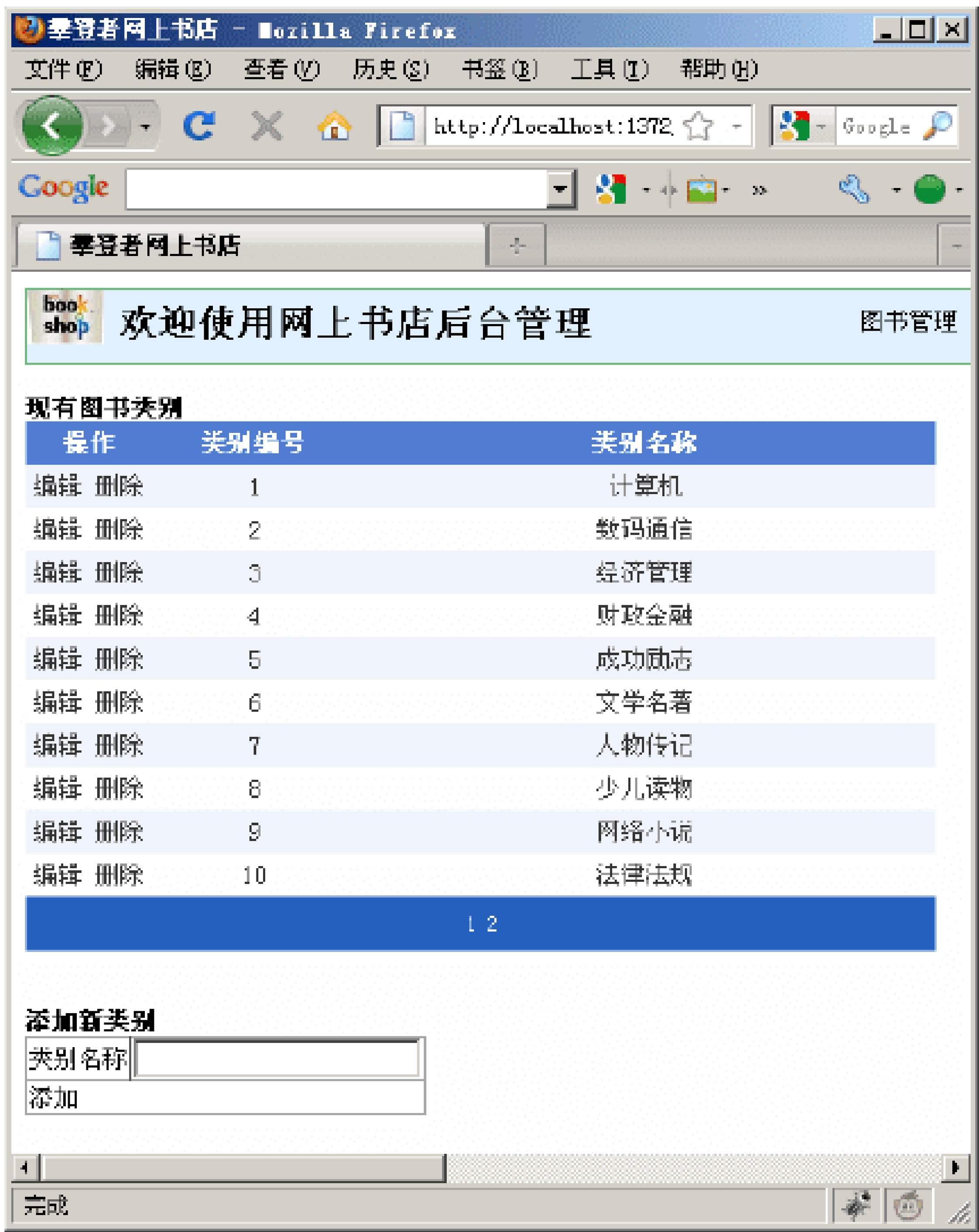


图 4.8 图书类别管理界面

4.3.5 图书管理

在图书管理页面中，管理员可以搜索和查看图书信息，并可以单击相应链接对图书进行编辑和删除操作。这个页面与前台中的高级搜索页面有一些相同之处，但是二者在图书列表中对图书的可用操作不同，前台可以将图书添加到购物车和查看图书详情，而后台则可以删除图书和编辑图书信息。

（1）以 Admin.master 作为母版页添加一个新页面 BookAdmin.aspx。

（2）在 BookAdmin.aspx 页面中放置多个 TextBox 控件以输入搜索条件。

```
<div class="dashedborder">
标题：<asp:TextBox ID="bookTitle" runat="server"/>
 作者：<asp:TextBox ID="author" runat="server"/>
 出版社：<asp:TextBox ID="publisher" runat="server"/>
<a  style="margin-left:150px; text-decoration:underline; "href=
"BookEdit.aspx?id=-1">添加新书</a><br />
价格范围：<asp:TextBox ID="price1" runat="server"></asp:TextBox>至<asp:
TextBox ID="price2" runat="server"></asp:TextBox>
 
出版日期：<asp:TextBox ID="date1" runat="server"></asp:TextBox>至<asp:
TextBox ID="date2" runat="server"></asp:TextBox>
    <asp:Button ID="Button1" runat="server" Text="搜索" Width="80"
      onclick="Button1 Click" />
</div>
```

（3）在 BookAdmin.aspx 页面中放置一个 Repeater 控件和一个 AspNetPager 控件以分页形式显示图书信息。在 Repeater 控件中除显示图书信息外，还有两个链接：删除图书和编辑图书。

```
<asp:Repeater ID="Repeater1" runat="server" onitemcommand="Repeater1
ItemCommand" >
```

```
<HeaderTemplate><table></HeaderTemplate>
<ItemTemplate>
<tr>
<td>
<%--将图书封面作为一个超链接--%>
<a href="BookEdit.aspx?id=<%#Eval("BookID")%>"> <img src="../BookImages/<
%#Eval ("SmallImage") %>" alt="<%#Eval("BookTitle") %>" style="width:
100px;" /></a>
</td>
<td>
<h3> <%#Eval("BookTitle") %> </h3>                <%--图书标题--%>
<%--以下内容显示了图书的作者、定价、会员价、折扣信息--%>
作者：<%#Convert.ToString(Eval("Author1")) + Convert.ToString(Eval
 ("Author2")) + Convert.ToString(Eval("Author3"))%>|
出版社：<%#Eval("PublisherName") %>|  出版日期：<%#Eval("PublishDate") %>
<br /><br />
定价：<span style="color:Blue;"> <%#Eval("Price","{0:C}") %></span>  |
会员价：<span style="color:red;"><%#Eval("RealPrice","{0:C}")%></span>
 折扣：<%#Eval("Discount")%><br /><br />
<%--以下为修改图书和删除图书两个链接--%>
<span style="margin-left:50px;">
<a href="BookEdit.aspx?id=<%#Eval("BookID")%>">修改图书信息</a>
</span>
<asp:LinkButton ID="Button1" runat="server" Text="删除图书" CommandName=
"delete"
CommandArgument='<%#Eval("BookID") %>' OnClientClick="return
confirmDelete(this);" />
</td>
</tr>
</ItemTemplate>
<FooterTemplate></table></FooterTemplate>
</asp:Repeater>
<%--分页控件--%>
<webdiyer:AspNetPager ID="AspNetPager1" runat="server"
   onpagechanged="AspNetPager1 PageChanged">
</webdiyer:AspNetPager>
```

（4）在“搜索”按钮的 Click 事件中及 AspNetPager 控件的 PageChanged 事件中，编写代码实现搜索结果的分页显示。具体代码与前台的高级搜索页面类似，此处省略。

（5）在 Repeater 控件的 ItemCommand 事件中，编写代码删除所选择的图书。

```
protected void Repeater1_ItemCommand(object source, RepeaterCommand-
EventArgs e)
{
   if (e.CommandName == "delete")                //如果命令名为删除
   {
      int id = Convert.ToInt32(e.CommandArgument);//获取命令参数（图书 ID）
      //构建 delete 语句，并执行
      string sql = "delete from Book where BookID=@id";
      SqlHelper db = new SqlHelper();
      var Command = db.GetSqlStringCommond(sql);
      db.AddInParameter(Command, "@id", System.Data.DbType.Int32, id);
      db.ExecuteNonQuery(Command);
      bindData();                                //重新绑定数据
   }
}
```

（6）BookAdmin.aspx 页面运行界面如图 4.9 所示。



图 4.9　图书管理页面

4.3.6　图书详情编辑设计思路

在图书详情编辑页面中，管理员可以修改图书的所有明细信息，如标题、作者、目录、内容简介、封面等。图书详情涉及的信息很多，而且数据量较大，例如，图书的目录和内容简介都会包含许多文字，图书的大封面和小封面图片也会占据页面很大的空间，在一个页面上不足以显示全部图书信息。

为了显示和编辑图书信息的方便，可以使用类似于标签页控件 Tab 的控件组成图书信息。把整个图书信息拆分成几个部分，在页面上任一时刻只显示其中某一部分内容，可以方便地在几个部分之间进行切换。在 ASP.NET 标准服务器控件中不包含 Tab 控件，但是提供了一个类似的控件叫做 MultiView。

提示：网上有许多第三方控件库实现了 Tab 控件，也可以使用这些第三方控件完成图书详情编辑页面的布局。

正如其名称所示，MultiView 控件可以在一个页面上显示多个视图。MultiView 中包含多个视图，每个视图都有一个索引，任一时刻只有一个视图可见，这个可见视图称为活动视图。可以通过设置 MultiView 控件的 ActiveViewIndex 属性在各个视图之间进行切换。

由于图书详情包含的数据量太大，如果把所有图书信息放在同一个页面中，即使使用了 MultiView 控件，页面代码也会变得冗长、晦涩、不易维护。为了缩小模块规模，可以把图书信息拆分成几个部分，每个部分分别用一个用户控件显示和编辑。按照这种思路，可以把图书信息拆分成 3 部分：图书基本信息、图书封面图片和图书所属类别，并且分别用 3 个用户控件完成各个部分信息的显示和编辑。

4.3.7　图书基本信息编辑控件

在图书基本信息控件中，可以显示和修改一个特定图书除封面图片和类别以外的所有信息。在这些信息中有两个属性比较特殊：图书目录和内容简介，这两个属性包含文字量

较大，而且希望包含格式控件符号，从而可以设置字体、颜色、边框等。可以使用 HTML 编辑器来满足这种需求。HTML 编辑器界面就像一个精简版的 Word 软件，允许用户以所见即所得的方式编辑文字，并自动生成对应的 HTML 代码。网上有多种 HTML 编辑器，本项目使用 FreeTextBox 控件（控件网址为 http://freetextbox.com/）。

（1）添加一个 BookEditControl 用户控件。

（2）下载 FreeTextBox 控件并将其添加到 Visual Studio 工具箱中。

（3）在 BookEditControl 用户控件中添加一个 SqlDataSource 控件，从 Book 表读取数据，并生成 UPDATE 和 DELETE 语句。要注意从代码视图中手工删除 SqlDataSource 控件自动生成的 Update 语句中设置 SmallImage 和 BigImage 的代码。

```
<asp:SqlDataSource ID="bookDataSource" runat="server"
   ConnectionString="<%$ ConnectionStrings:database %>"
   SelectCommand="SELECT * FROM [Book] WHERE ([BookID] = @BookID)"
   DeleteCommand="DELETE FROM [Book] WHERE [BookID] = @BookID"
   InsertCommand="INSERT INTO [Book] ([BookTitle], [Author1], [Author2],
   [Author3], [PublisherID], [Price], [Discount], [PublishDate],
   [Description], [Contents], [SaleSum], [ClickCount]) VALUES (@BookTitle,
   @Author1, @Author2, @Author3, @PublisherID, @Price, @Discount,
   @PublishDate, @Description, @Contents, @SaleSum, @ClickCount)"
   UpdateCommand="UPDATE [Book] SET [BookTitle] = @BookTitle, [Author1] =
   @Author1, [Author2] = @Author2, [Author3] = @Author3, [PublisherID] =
   @PublisherID, [Price] = @Price, [Discount] = @Discount,  [PublishDate]
   = @PublishDate, [Description] = @Description, [Contents] = @Contents,
   [SaleSum] = @SaleSum, [ClickCount] = @ClickCount WHERE [BookID] =
   @BookID">
   <DeleteParameters>
      <asp:Parameter Name="BookID" Type="Int32" />
   </DeleteParameters>
   <InsertParameters>
      <asp:Parameter Name="BookTitle" Type="String" />
      <asp:Parameter Name="Author1" Type="String" />
      <asp:Parameter Name="Author2" Type="String" />
      <asp:Parameter Name="Author3" Type="String" />
      <asp:Parameter Name="PublisherID" Type="Int32" />
      <asp:Parameter Name="Price" Type="Double" />
      <asp:Parameter Name="Discount" Type="Double" />
      <asp:Parameter Name="PublishDate" Type="DateTime" />
      <asp:Parameter Name="Description" Type="String" />
      <asp:Parameter Name="Contents" Type="String" />
      <asp:Parameter Name="SaleSum" Type="Int32" />
      <asp:Parameter Name="ClickCount" Type="Int32" />
   </InsertParameters>
   <SelectParameters>
      <asp:QueryStringParameter Name="BookID" QueryStringField="id"
      Type="Int32" />
   </SelectParameters>
   <UpdateParameters>
      <asp:Parameter Name="BookTitle" Type="String" />
      <asp:Parameter Name="Author1" Type="String" />
      <asp:Parameter Name="Author2" Type="String" />
      <asp:Parameter Name="Author3" Type="String" />
      <asp:Parameter Name="PublisherID" Type="Int32" />
      <asp:Parameter Name="Price" Type="Double" />
      <asp:Parameter Name="Discount" Type="Double" />
      <asp:Parameter Name="PublishDate" Type="DateTime" />
      <asp:Parameter Name="Description" Type="String" />
```

```
        <asp:Parameter Name="Contents" Type="String" />
        <asp:Parameter Name="SaleSum" Type="Int32" />
        <asp:Parameter Name="ClickCount" Type="Int32" />
        <asp:Parameter Name="BookID" Type="Int32" />
    </UpdateParameters>
</asp:SqlDataSource>
```

提示： 由于此控件不修改图书封面图片，所以要从代码视图中手工删除 SqlDataSource 控件自动生成的 Update 语句中设置 SmallImage 和 BigImage 的代码，否则在更新数据时就会把封面图片清空。

（4）在 BookEditControl 用户控件中添加一个 FormView 控件，配置 FormView 的 ItemTemplate 以显示和编辑图书信息，注意其中使用 FreeTextBox 控件编辑图书简介和图书目录。

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="BookID"
    DataSourceID="bookDataSource" DefaultMode="Edit"
oniteminserted="FormView1 ItemInserted">
    <EditItemTemplate>
       图书编号:
        <asp:Label ID="BookIDLabel1" runat="server" Text='<%# Eval
         ("BookID") %>' />
        <br />
        题目:<asp:TextBox ID="BookTitleTextBox" runat="server"  Text='<%#
        Bind("BookTitle") %>' Width="200" />
    作者 1:<asp:TextBox ID="Author1TextBox" runat="server" Text='<%#
    Bind("Author1") %>' />
    作者 2:<asp:TextBox ID="Author2TextBox" runat="server" Text='<%#
    Bind("Author2") %>' />
    作者 3: <asp:TextBox ID="Author3TextBox" runat="server" Text='<%#
    Bind("Author3") %>' />
        <br />
        出版社:
        <asp:DropDownList runat="server" ID="publisherList" Width="200"
        DataValueField="PublisherID" DataTextField="PublisherName"
         SelectedValue='<%# Bind("PublisherID") %>' DataSourceID="public-
         sherDataSource" />
        定价:<asp:TextBox ID="PriceTextBox" runat="server" Text='<%#
        Bind("Price") %>' />
    折扣:<asp:TextBox ID="DiscountTextBox" runat="server" Text='<%#
    Bind("Discount") %>' />
        会员价格: <asp:Label ID="RealPriceTextBox" runat="server"  Text='<%#
        Eval("RealPrice") %>' />
        <br />
        出版日期: <asp:TextBox ID="PublishDateTextBox" runat="server" Text=
        '<%# Bind("PublishDate") %>' />
         总销售量: <asp:TextBox ID="SaleSumTextBox" runat="server" Text='<%#
         Bind("SaleSum") %>' />
        总单击量: <asp:TextBox ID="ClickCountTextBox" runat="server"  Text=
        '<%# Bind("ClickCount") %>' />
         <asp:Button ID="UpdateButton" runat="server" CausesValidation="True"
            CommandName="Update" Text="保存" />
         <asp:Button ID="UpdateCancelButton" runat="server"
            CausesValidation="False" CommandName="Cancel" Text="取消" />
        <br />
        图书简介:<br />
```

```
        <FTB:FreeTextBox ID="descriptionTextBox" runat="server" Text='<%#
        Bind("Description") %>'
         Width="900" Height="200"> </FTB:FreeTextBox>  <br />
        目录:<br />
       <FTB:FreeTextBox ID="ContentTextBox" runat="server" Text='<%#
       Bind("Contents") %>'
       Width="900" Height="200">  </FTB:FreeTextBox><br />
    </EditItemTemplate>
    <InsertItemTemplate>
         图书编号:
        <asp:Label ID="BookIDLabel1" runat="server" Text='<%# Eval("Book-
        ID") %>' />
        <br />
        题目:<asp:TextBox ID="BookTitleTextBox" runat="server"  Text='<%#
        Bind("BookTitle") %>' Width="200" />
    作者 1:<asp:TextBox ID="Author1TextBox" runat="server" Text='<%#
    Bind("Author1") %>' />
    作者 2:<asp:TextBox ID="Author2TextBox" runat="server" Text='<%#
    Bind("Author2") %>' />
    作者 3: <asp:TextBox ID="Author3TextBox" runat="server" Text='<%#
    Bind("Author3") %>' />
        <br />
        出版社:
        <asp:DropDownList runat="server" ID="publisherList" Width="200"
        DataValueField="PublisherID" DataTextField="PublisherName"
        SelectedValue='<%# Bind("PublisherID") %>' DataSourceID="public-
        sherDataSource" />
        定价:<asp:TextBox ID="PriceTextBox" runat="server" Text='<%# Bind
        ("Price") %>' />
    折扣:<asp:TextBox ID="DiscountTextBox" runat="server" Text='<%#
    Bind("Discount") %>' />
        <br />
        出版日期: <asp:TextBox ID="PublishDateTextBox" runat="server" Text=
        '<%# Bind("PublishDate") %>' />
         总销售量: <asp:TextBox ID="SaleSumTextBox" runat="server" Text='<%#
         Bind("SaleSum") %>' />
        总单击量: <asp:TextBox ID="ClickCountTextBox" runat="server"  Text=
        '<%# Bind("ClickCount") %>' />
         <asp:Button ID="UpdateButton" runat="server" CausesValidation=
         "True"
            CommandName="Insert" Text="添加" />
         <asp:Button ID="UpdateCancelButton" runat="server"
            CausesValidation="False" CommandName="Cancel" Text="取消" />
        <br />
        图书简介:<br />
        <FTB:FreeTextBox ID="descriptionTextBox" runat="server" Text='<%#
        Bind("Description") %>'
         Width="900" Height="200"> </FTB:FreeTextBox>  <br />
        目录:<br />
       <FTB:FreeTextBox ID="ContentTextBox" runat="server" Text='<%# Bind
       ("Contents") %>'
       Width="900" Height="200">  </FTB:FreeTextBox><br />
    </InsertItemTemplate>
</asp:FormView>
```

（5）在 BookEditControl 用户控件中可放置一个 SqlDataSource 控件，从 Publisher 表中检索数据，并绑定到 FormView 控件 ItemTemplate 中的 DropDownList 控件。

```
<asp:SqlDataSource ID="publisherDataSource" runat="server"
   ConnectionString="<%$ ConnectionStrings:database %>"
   SelectCommand="SELECT [PublisherID], [PublisherName] FROM [Publisher]
   "></asp:SqlDataSource>
```

（6）在 BookEditControl 用户控件的 Page_Load 事件中，根据 QueryString 传递的图书编号设置 FormView 的格式。如果图书编号为大于 0，则以编辑模式显示此编号对应的图书，如果图书编号为–1，则说明需要添加新图书，将 FormView 更改为插入模式。

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        if (Request.QueryString["id"] == "-1")
           FormView1.ChangeMode(FormViewMode.Insert);
    }
}
```

（7）当新图书信息首次保存时，将页面跳转到 BookEdit 页面，并将新图书的 ID 作为 QueryString 参数传递过去。为此，需要在 FormView 的 ItemInserted 事件中编写以下代码。

```
protected void FormView1_ItemInserted(object sender, FormViewInserted-
EventArgs e)
{
    string sql = "select max(BookID) from Book";        //得到最新图书 ID
    SqlHelper db = new SqlHelper();
    var command = db.GetSqlStringCommond(sql);
    int n = Convert.ToInt32(db.ExecuteScalar(command));
    Response.Redirect("BookEdit.aspx?id="+n);           //重新跳转到此页面
}
```

（8）添加一个新页面以测试 BookEditControl 用户控件，运行界面如图 4.10 所示。

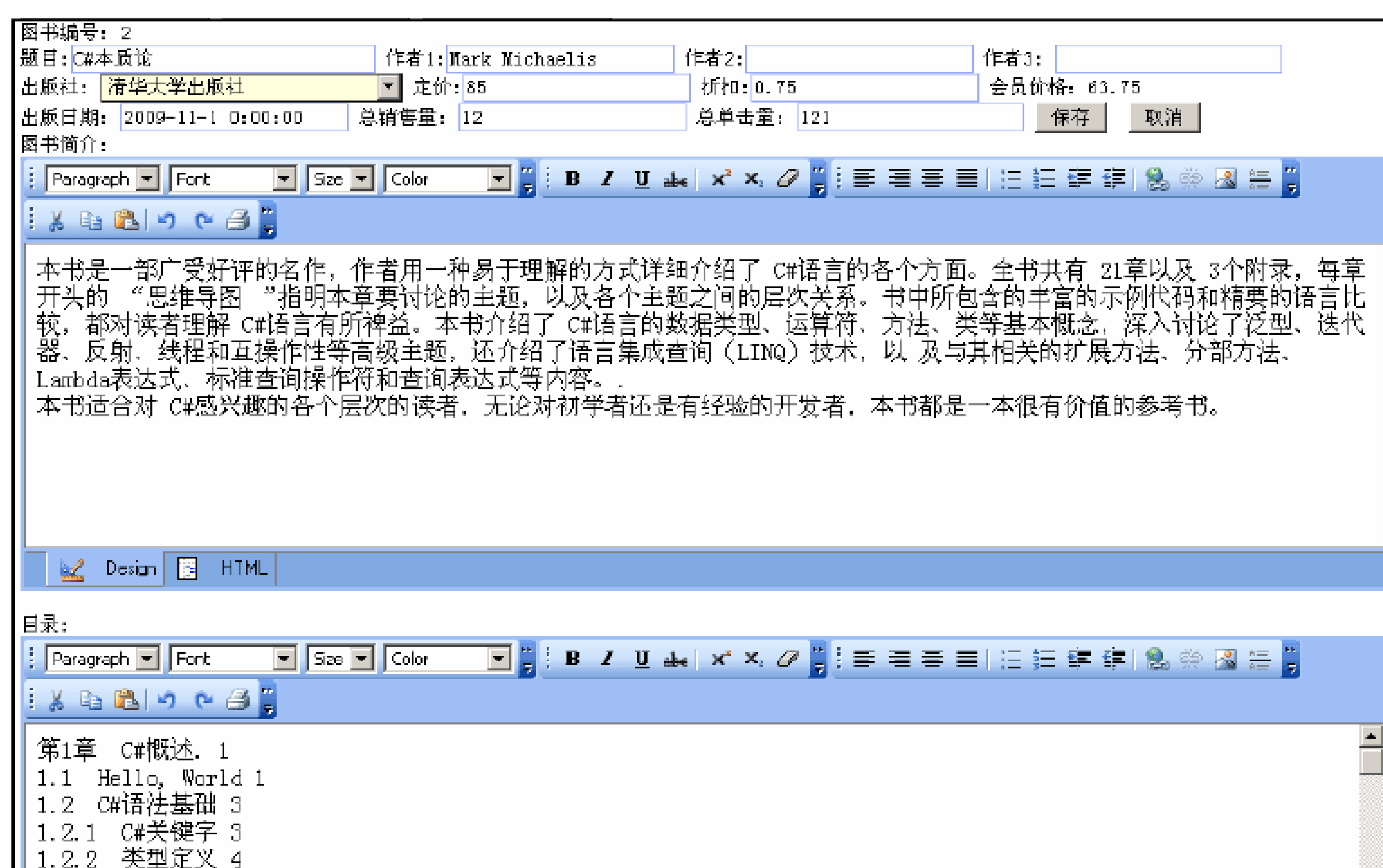


图 4.10　图书基本信息编辑控件

4.3.8　图书封面编辑控件

在图书封面控件中，可以显示一个特定图书的现有的小封面图片和大封面图片，可以

上传新的图书封面替换原有的封面。在上传文件时，要注意把文件重命名。这是因为数据库中会有许多书，如果按照上传时的文件名保存文件，则很可能造成文件名重名，从而覆盖早期的文件。在文件上传时，应该按照一种规则将文件重命名，以确保每个文件名的唯一性。通常用的重命名规则包括用数据库主键、当前日期时间，或者二者结合作为文件名。

（1）添加一个新用户控件 BookCoverEdit.ascx。

（2）在 BookCoverEdit 控件上放置一个 FormView 和 SqlDataSource，用于检索图书封面信息并显示。图书编号通过 QueryString 传递给页面，要给 SqlDataSource 添加一个 QueryString 参数。

```
<h3>图书封面</h3>
<div style="float:left;">
<h3>原有封面</h3>
<asp:FormView ID="FormView1" runat="server" DataSourceID="SqlDataSource1" >
<ItemTemplate>
小封面<br />
<img src="../BookImages/<%#Eval("SmallImage")%>"  style="width:100px;"
alt="<%#Eval("BookTitle")%>"/><br />
大封面<br />
<img src="../BookImages/<%#Eval("BigImage")%>" style="width:200px;" alt=
"<%#Eval("BookTitle")%>" /><br />
</ItemTemplate>
</asp:FormView>
</div>
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
   ConnectionString="<%$ ConnectionStrings:database %>"
   SelectCommand="SELECT [BookID], [BookTitle], [BigImage], [SmallImage]
   FROM [Book] WHERE ([BookID] = @BookID)">
   <SelectParameters>
      <asp:QueryStringParameter Name="BookID" QueryStringField="id" Type=
      "Int32" />
   </SelectParameters>
</asp:SqlDataSource>
```

（3）在 BookCoverEdit 控件上放置两个 FileUpload 控件和两个 Button 控件，以上传新的图书封面。

```
<div >
<h3>上传新封面</h3>
小封面<asp:FileUpload ID="smallPicture" runat="server"  />
   <asp:Button ID="Button1" runat="server" Text="上传并替换" onclick=
   "Button1_Click" />
<br />
大封面<asp:FileUpload ID="bigPicture" runat="server"  />
<asp:Button ID="Button2" runat="server" Text="上传并替换" onclick="Button2_
Click" /><br />
</div>
```

（4）在 Page_Load 事件中，根据 QueryString 所传递的图书 ID 的合法性，设置两个上传按钮的可用性。

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
   {
      string id = Request.QueryString["id"];
```

```
        if (string.IsNullOrEmpty(id) || id == "-1")
        {
            Button1.Enabled = false;
            Button2.Enabled = false;
        }
    }
}
```

（5）在两个上传按钮的 Click 事件中，将新的封面图片文件上传到服务器，并删除原有图片文件。在上传文件时，把文件进行重命名，命名规则为：图书主键+s（表示 small）或 b（表示 big）+扩展名。

```
protected void Button1 Click(object sender, EventArgs e)
{
    updateBookCover(smallPicture, false);
}
protected void Button2 Click(object sender, EventArgs e)
{
    updateBookCover(bigPicture, true);
}
/// <summary>
/// 更新图书封面
/// </summary>
/// <param name="file">包含封面图片的文件上传控件</param>
/// <param name="big">是否大封面</param>
private void updateBookCover(FileUpload file, bool big)
{
    if (!file.HasFile) return;
    //取得原来的封面文件并删除
    SqlHelper db = new SqlHelper();
    string sql = string.Format("select {0} from Book where BookID={1}"
        , big?"BigImage":"SmallImage" ,Request.QueryString["id"]);
    DbCommand command = db.GetSqlStringCommond(sql);
    string old = db.ExecuteScalar(command).ToString();
    old = Server.MapPath("~/BookImages/" + old);
    if(System.IO.File.Exists(old))
        System.IO.File.Delete(old);
    //得到新封面文件名,命名规则为：图书 ID + s 或 b + 文件扩展名
    string newName = Request.QueryString["id"]
        + (big ? "b" : "s")
        + System.IO.Path.GetExtension(file.FileName);
    file.SaveAs(Server.MapPath("~/BookImages/" + newName));
    //更新数据库中图书封面数据
    sql="update Book set " + (big?"BigImage":"SmallImage")
        +"=@img where BookID=@id";
    command = db.GetSqlStringCommond(sql);
    db.AddInParameter(command, "@id", DbType.Int32,
    int.Parse(Request.QueryString["id"]));
    db.AddInParameter(command, "@img", DbType.String, newName);
    db.ExecuteNonQuery(command);
    FormView1.DataBind();
}
```

（6）创建一个测试页面以测试 BookCoverEdit 控件，运行界面如图 4.11 所示。



图 4.11　图书封面编辑控件

4.3.9　图书类别编辑控件

在图书类别编辑控件中，管理员可以编辑一个特定图书的所属类别。一个图书可以同时属于零个或者多个类别，图书和类别之间是多对多的关系。由于图书类别涉及的数据量很小，使用这个控件的用户很少（只有管理员才能用），所以这个页面可以使用 GridView 自带的数据编辑和删除功能。

（1）添加一个用户控件 BookCategoryControl.ascx。

（2）在用户控件上放置一个 SqlDataSource 控件，从检索 BookCategory 表中检索数据，并自动生成 UPDATE 和 DELETE 语句。注意为 SqlDataSource 控件添加 QueryString 参数。

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%$ ConnectionStrings:database %>"
    DeleteCommand="DELETE FROM [BookCategory] WHERE [BookID] = @BookID AND
    [CategoryID] = @CategoryID"
    InsertCommand="INSERT INTO [BookCategory] ([BookID], [CategoryID])
    VALUES (@BookID, @CategoryID)"
    SelectCommand="SELECT [BookID], [CategoryID] FROM [BookCategory] WHERE
    ([BookID] = @BookID)">
    <DeleteParameters>
        <asp:Parameter Name="BookID" Type="Int32" />
        <asp:Parameter Name="CategoryID" Type="Int32" />
    </DeleteParameters>
    <InsertParameters>
        <asp:Parameter Name="BookID" Type="Int32" />
        <asp:Parameter Name="CategoryID" Type="Int32" />
    </InsertParameters>
    <SelectParameters>
        <asp:QueryStringParameter Name="BookID" QueryStringField="id"
        Type="Int32" />
    </SelectParameters>
</asp:SqlDataSource>
```

（3）在用户控件上添加一个 GridView 控件，将其数据源设置为 SqlDataSource1，设置 GridView 中的各个字段标题。将 GridView 的所属类别一列设置为模板列，用 DropDownList 控件显示图书类别。

```
<h3>图书所属类别</h3>
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
   DataKeyNames="BookID,CategoryID" DataSourceID="SqlDataSource1">
   <Columns>
      <asp:CommandField ShowDeleteButton="True" DeleteText="删除"  />
      <asp:BoundField DataField="BookID" HeaderText="图书编号" ReadOnly=
      "True"
         ItemStyle-Width="100" ItemStyle-HorizontalAlign="Center" />
         <asp:TemplateField ItemStyle-Width="300" ItemStyle-Horizon-
         talAlign="Center" HeaderText="所属类别" >
         <ItemTemplate >
         <asp:DropDownList runat="server" ID="categoryList" DataSour-
         ceID="SqlDataSource2"
         DataValueField="categoryID" DataTextField="CategoryName"
         SelectedValue='<%#Eval("CategoryID") %>' />
         </ItemTemplate>
         </asp:TemplateField>
   </Columns>
</asp:GridView>
```

（4）在用户控件中添加另外一个 SqlDataSource，从 Category 表中检索数据，为 GridView 控件模板列中的 DropDownList 控件提供数据源。

```
<asp:SqlDataSource ID="SqlDataSource2" runat="server"
   ConnectionString="<%$ ConnectionStrings:database %>"
   SelectCommand="SELECT * FROM [Category]"></asp:SqlDataSource>
```

（5）在用户控件底部放置一个 DropDownList 和一个 Button，以添加新的类别。

```
<b>添加新的类别</b><br />
<asp:DropDownList runat="server" ID="categoryList" DataSourceID=
"SqlDataSource2"
       DataValueField="categoryID" DataTextField="CategoryName" />
<asp:Button ID="button1" runat="server" Text="添加" Width="80"
   onclick="button1_Click" />
```

（6）在“添加”按钮的 Click 事件中完成数据库操作，为图书添加所选类别。

```
protected void button1 Click(object sender, EventArgs e)
{
   string bookid = Request.QueryString["id"];
   string category = categoryList.SelectedValue;
   string sql = "insert into BookCategory (BookID,CategoryID)
   values(@bid,@cid)";
   SqlHelper db = new SqlHelper();
   System.Data.Common.DbCommand command=db.GetSqlStringCommond(sql);
   db.AddInParameter(command, "@bid", System.Data.DbType.Int32, bookid);
   db.AddInParameter(command, "@cid", System.Data.DbType.Int32, int.Parse
   (category));
   db.ExecuteNonQuery(command);
   GridView1.DataBind();
}
```

（7）添加一个测试页面以测试 BookCategoryControl 用户控件，运行界面如图 4.12 所示。



图 4.12　图书类别编辑用户控件

4.3.10　图书编辑页面

在图书编辑页面 BookEdit.aspx 中，以 MultiView 控件在 3 个视图中显示前面所创建的 3 个用户控件：基本信息编辑控件、图书封面编辑控件、图书类别编辑控件。用户可以在 3 个视图之间进行切换，查看并编辑相关信息。BookEdit.aspx 页面代码如下 L：

```
//BookEdit.aspx
<%--注册 1 个自定义控件（HTML 文档编辑器）和 3 个用户控件--%>
<%@ Register Assembly="FreeTextBox" Namespace="FreeTextBoxControls"
TagPrefix="FTB" %>
<%@ Register src="BookEditControl.ascx" tagname="BookEditControl"
tagprefix="uc1" %>
<%@ Register src="BookCoverEdit.ascx" tagname="BookCoverEdit"
tagprefix="uc2" %>
<%@ Register src="BookCategoryControl.ascx" tagname="BookCategoryControl"
tagprefix="uc3" %>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
    <asp:Button ID="Button1" runat="server" Text="修改图书基本信息"
        onclick="Button1_Click" />
    <asp:Button ID="Button2"  runat="server" Text="修改图书封面图片"
        onclick="Button2_Click" />
    <asp:Button ID="Button3" runat="server" Text="修改图书所属类别"
        onclick="Button3_Click" />
<asp:MultiView runat="server" ID="multiview" ActiveViewIndex="0" >
<%--第一个视图，显示图书编辑控件--%>
<asp:View runat="server" ID="view1">
<uc1:BookEditControl ID="control1" runat="server" />
</asp:View>
<%--第二个视图，显示图书封面编辑控件--%>
 <asp:View runat="server" ID="view2">
 <uc2:BookCoverEdit id="control2" runat="server" />
```

```
 </asp:View>
<%--第三个视图，显示图书类别编辑控件--%>
 <asp:View runat="server" ID="view3">
 <uc3:BookCategoryControl id="control3" runat="server" />
 </asp:View>
</asp:MultiView>
</asp:Content>
//BookEdit.aspx.cs
public partial class BookEdit : System.Web.UI.Page
{
    //单击 3 个按钮时，分别切换到 3 个视图
    protected void Button1 Click(object sender, EventArgs e)
    {
        multiview.ActiveViewIndex = 0;
    }
    protected void Button2_Click(object sender, EventArgs e)
    {
        multiview.ActiveViewIndex = 1;
    }
    protected void Button3_Click(object sender, EventArgs e)
    {
        multiview.ActiveViewIndex = 2;
    }
}
```

图书编辑页面 BookEdit.aspx 的运行界面，如图 4.13 所示。

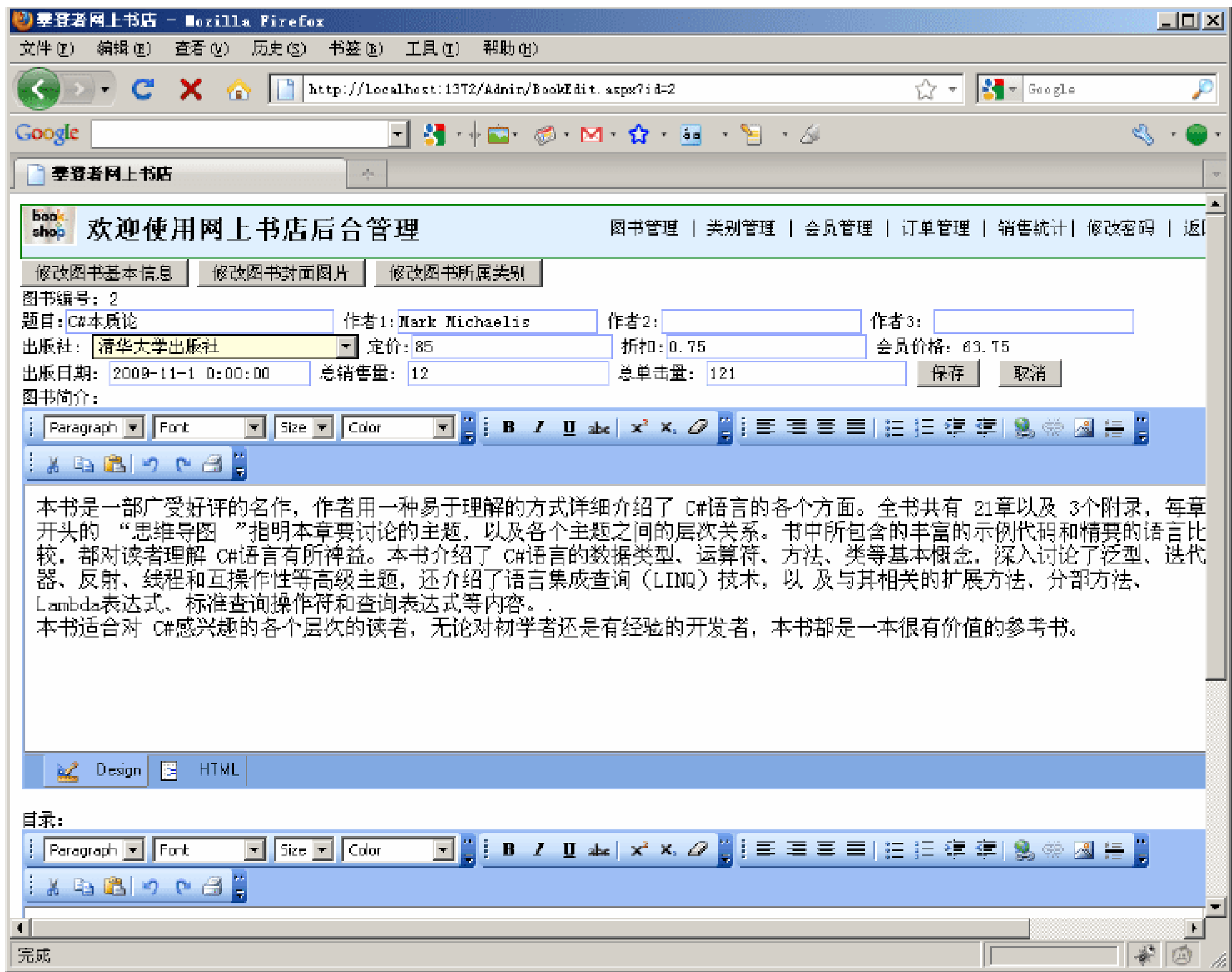


图 4.13　图书编辑页面

4.4　小　　结

本章综合利用前面几章所介绍的知识开发了一个网上书店，主要功能包括图书浏览、图书搜索、购物车、图书编辑等。网上书店是一种典型的电子商务网站，熟练掌握这个案例对于巩固 ASP.NET 开发技术、增加项目经验、开发类似电子商务系统都有很大帮助。

第 5 章　规范的软件开发

软件不等于编码，要想成为一名合格的软件开发人员，除了要掌握过硬的编码技术以外，还需要熟悉规范的软件开发流程，能够按照软件工程思想完成软件开发全过程。本章将对三层结构、源码管理和软件测试 3 个主题进行介绍。

5.1　源码管理简介

在软件开发过程中，会产生大量的源码文件。这些源码文件分散于整个团队中，团队中很多成员都可能会不定期地修改源码文件中的任何一个。如何正确地管理好这么多源码文件，协调各个开发人员之间的工作，对于整个软件项目来说是一项重要工作。源码文件数量庞大，修改频繁，不可能手工维护这些文件，而必须依赖于源码管理软件。本节将以与 Visual Studio 紧密集成的微软的 Visual SourceSafe 软件为例，讲解源码管理各个任务的实现。

源码管理主要任务包含两方面：一是跟踪源码修改的整个过程，保存修改的历史记录，并可以获取历史记录的任一版本，可以对任意两个版本进行比较。二是实现多个开发人员协同工作，当多人修改同一文件时保证文件的一致性，避免多个修改之间相互覆盖。

开发人员在整个开发过程中都可能对源码进行修改，而新的修改有时会给现有的程序引入 bug，甚至有时所做修改完全是方向性的错误，需要重新确定方案重新编码。当这种情况发生时，开发人员希望能够将源码恢复成以前未修改的状态，或者对比修改前后的源码有何不同，从而较容易的定位 bug 所在位置。使用源码管理工具可以方便地完成这些任务。

在软件开发团队中，多个开发人员共同完成一个项目。如果两个开发人员张三和李四同时修改一个源代码文件 BookDAL.cs，张三修改了其中的一个方法 getBookByID，而李四则添加了另外一个新的方法 deleteBook，那么就会产生两个不同版本的 BookDAL，每个版本只包含 BookDAL 类的一部分功能，这给整个软件项目的管理带来不便，并很可能引发其他错误。

为了避免出现多个人员同时修改一个文件从而造成版本混乱的情况，应该对文件的修改进行控制，可以采取一种类似于操作系统中管理临界资源的方式管理源码文件，把所有源码文件集中放置在源码服务器上，任一时刻最多只能有一名开发人员修改某一文件。

开发人员张三在修改某一文件 BookDAL.cs 之前，要进行申请，如果这个文件当前没有人修改，则批准张三的申请，并将文件 BookDAL.cs 标记为正在被张三修改。如果张三申请修改 BookDAL.cs 时，这个文件已经被其他开发人员申请修改了，则拒绝张三的申请，张三不能修改此文件。开发人员张三在修改完 BookDAL.cs 以后，要释放对这个文件的修改权限，并重新将 BookDAL.cs 标记为没有人修改。

在以上过程中，开发人员申请并修改一个文件的过程称为“签出”，意为将源码文件从服务器取出来给开发人员。开发人员对文件修改完成以后，把文件交还给服务器并标记为当前无人修改的状态，这个过程称为“签入”。

5.2　使用 Visual SourceSafe 管理源码

Visual SourceSafe（简称 VSS）是微软推出的源码管理工具，当前最新版本是 VSS 2005。微软在 VSS 2005 以后又推出了用于管理团队开发的软件 Team Foundation Server（简称 TFS），TFS 除包含 VSS 中的源码管理功能之外，还包括其他更多团队开发的相关功能。由于 VSS 的小巧（VSS 2005 安装程序只有 105M）和 TFS 的庞大（TFS 2010 安装程序有 2G 多），VSS 的应用仍然非常普遍。

5.2.1　VSS 用户管理

就像 SQL Server 数据库一样，VSS 必须有合法的用户名和密码才能访问。VSS 的用户有两种不同的权限：只读权限和读写权限。VSS 安装后会自动添加一个名为 Admin 的管理员用户，拥有最高权限。使用 Admin 登录后可以再创建和修改其他用户。

VSS 2005 安装后，会在 Windows 程序菜单中生成两个菜单项 Microsoft Visual SourceSafe 和 Microsoft Visual SourceSafe Admin，为叙述方便，在本章中将前者称为 VSS 程序，将后者称为 VSS 管理程序。前者用于源码管理，如签入、签出、版本回滚和比较等，后者主要用于 VSS 配置，如添加用户、配置服务、管理数据库等。VSS 管理程序启动后主界面如图 5.1 所示。

在图 5.1 所示的 VSS 管理程序主界面中，显示了所有用户列表及状态。如果要添加用户，则从菜单中选择“用户”|“添加用户”命令，则弹出如图 5.2 所示的“添加用户”对话框。在其中输入用户名和密码，并根据情况设置是否只读，然后单击“确定”按钮即可。

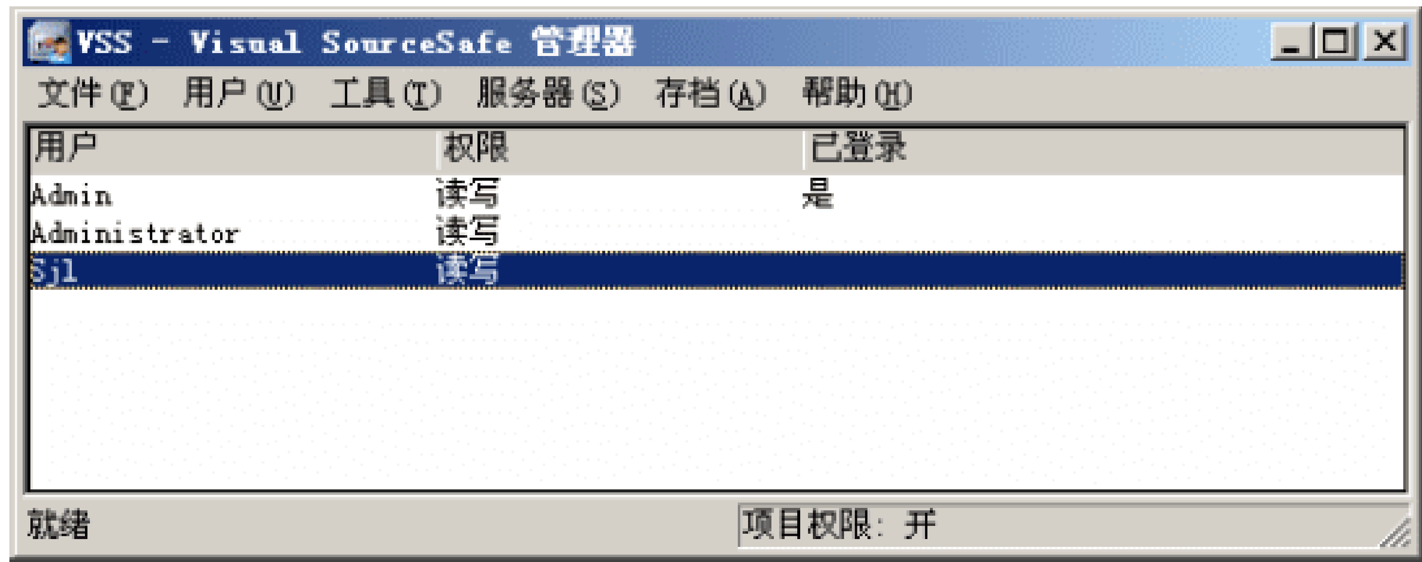


图 5.1　VSS 管理程序主界面



图 5.2　添加 VSS 用户

如果要删除 VSS 用户，则在图 5.1 所示的 VSS 管理程序主界面中选中一个用户，从菜单中选择“用户”|“删除用户”命令。如果要编辑用户信息，首先选中一个用户，再从 VSS 菜单中选择“用户”|“编辑用户”命令，会弹出“编辑用户”对话框，其界面与图 5.2 类似，在其中设置新用户名及读写权限即可。

5.2.2　管理 VSS 数据库

Visual SourceSafe 把所管理的文件保存到特定格式的 VSS 数据库中。用户可以创建、

删除 VSS 数据库，也可以选择当前所使用的 VSS 数据库。

（1）如果要创建 VSS 数据库，从 VSS 管理程序中选择“文件”|“新建数据库”命令，则弹出新建数据库向导对话框，如图 5.3 所示。

（2）单击“下一步”按钮，然后选择一个磁盘路径作为 VSS 数据库的保存路径，然后一直单击“下一步”按钮，直到出现如图 5.4 所示的选择 VSS 默认设置对话框。根据 VSS 签出签入方式的不同，VSS 有“锁定-修改-解锁”模型和“复制-修改-合并”模型两种不同的文件修改方式：

- ❑ 第一种模式下每个文件只允许一个用户签出，不会产生多个用户同时修改一个文件的情况，使用简单，不易出错。
- ❑ 第二种模式允许多个用户同时签出一个文件并进行编辑，然后将多个用户的修改进行合并。这种模式能够提高多用户并发性。



图 5.3　新建数据库向导

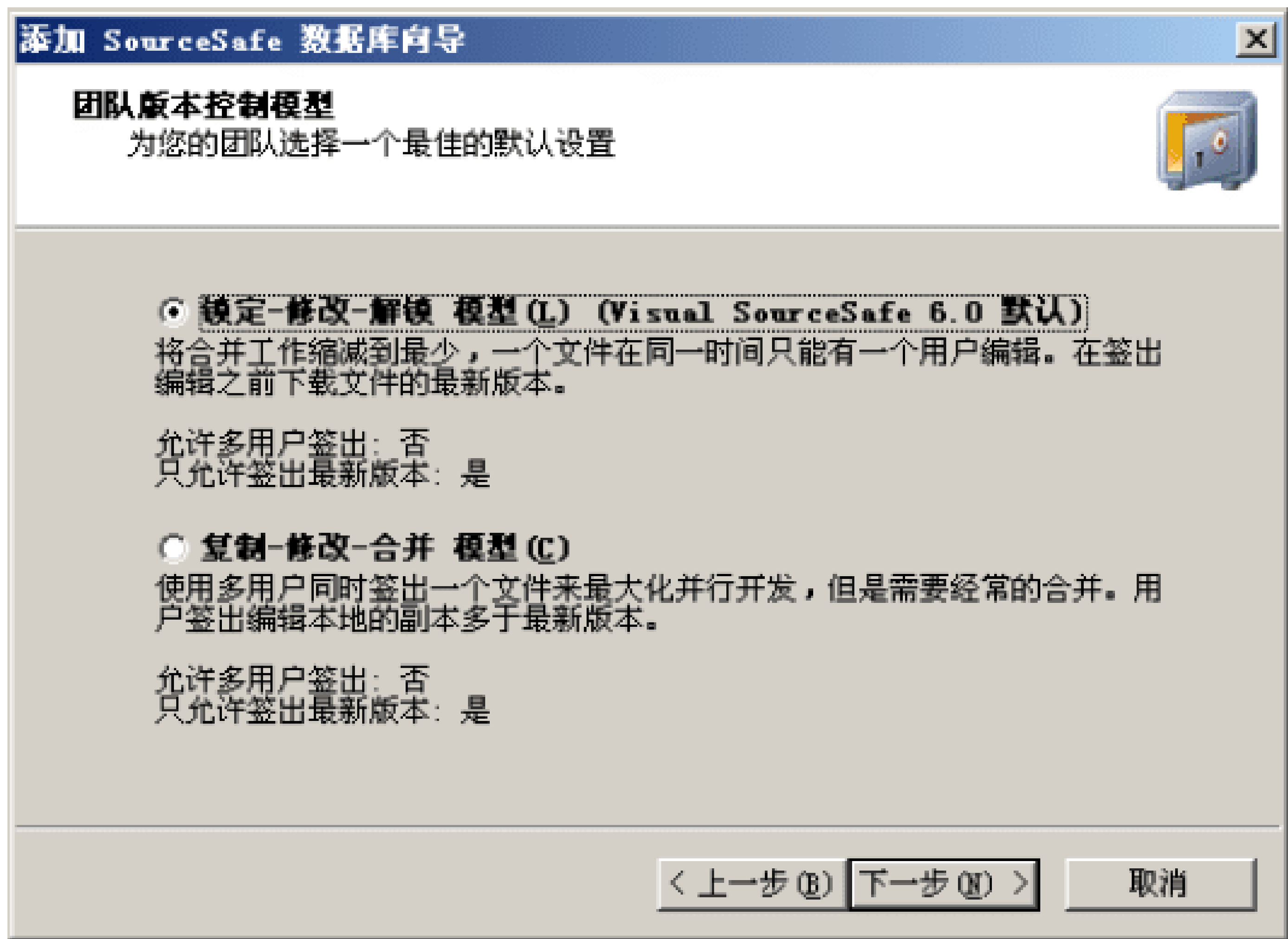


图 5.4　选择 VSS 默认设置

（3）如果 VSS 拥有多个数据库，可以通过 VSS 管理程序选择当前使用的数据库。具体操作步骤为从菜单中选择“文件”|“打开 SourceSafe 数据库”命令，则会弹出如图 5.5 所示对话框，在其中选择想要打开的数据库，单击“打开”按钮即可。

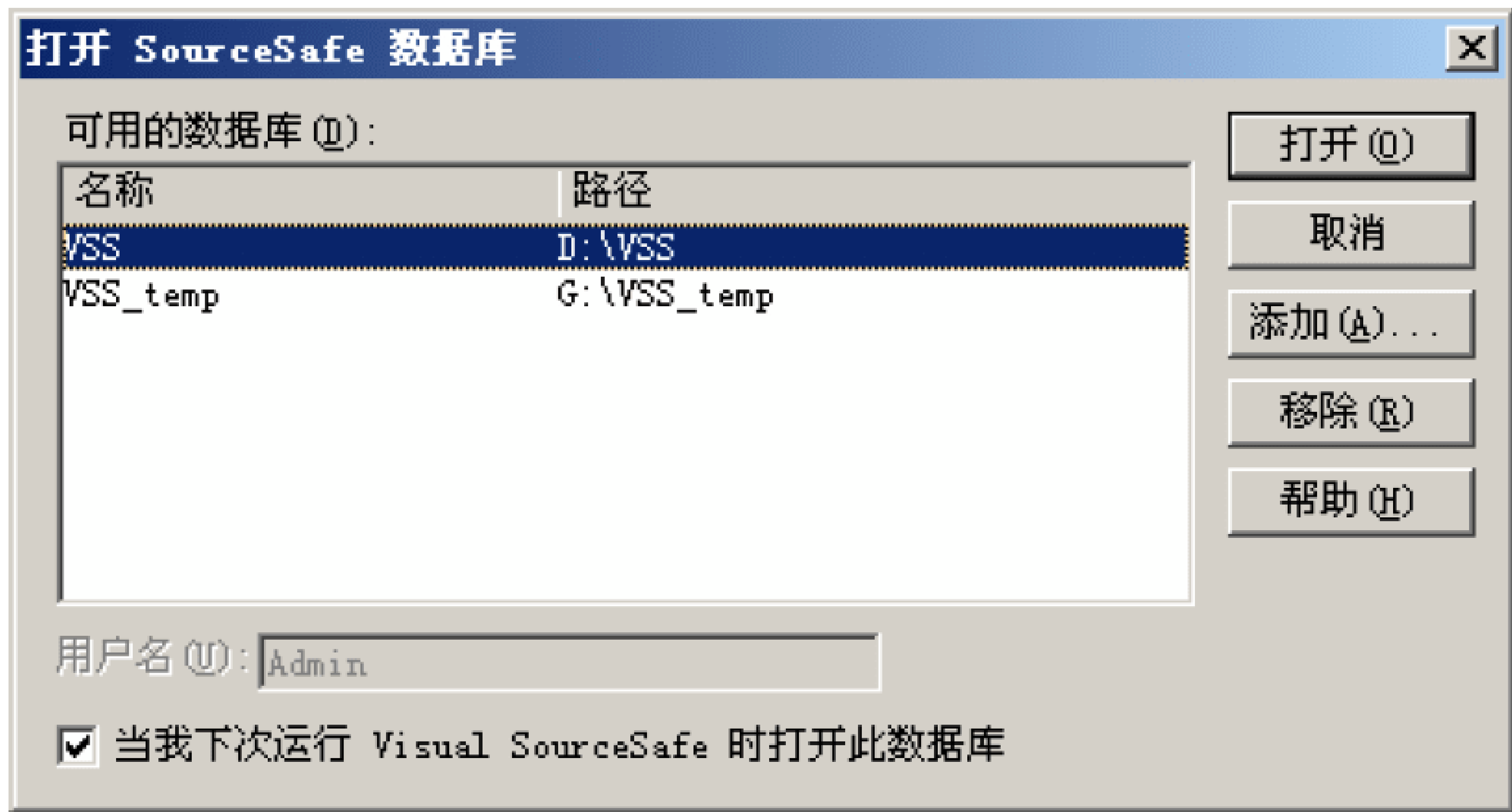


图 5.5　选择 VSS 数据库

（4）为了保证数据安全，还可以备份和恢复 VSS 数据库。要备份 VSS 数据库，则从 VSS 管理程序中选择“存档”|“将项目存档”命令，则弹出如图 5.6 所示“选择要存档的项目”

对话框。在其中选择要目录或者根目录下一个项目，单击“确定”按钮，则进入图 5.7 所示的“存档向导”对话框。

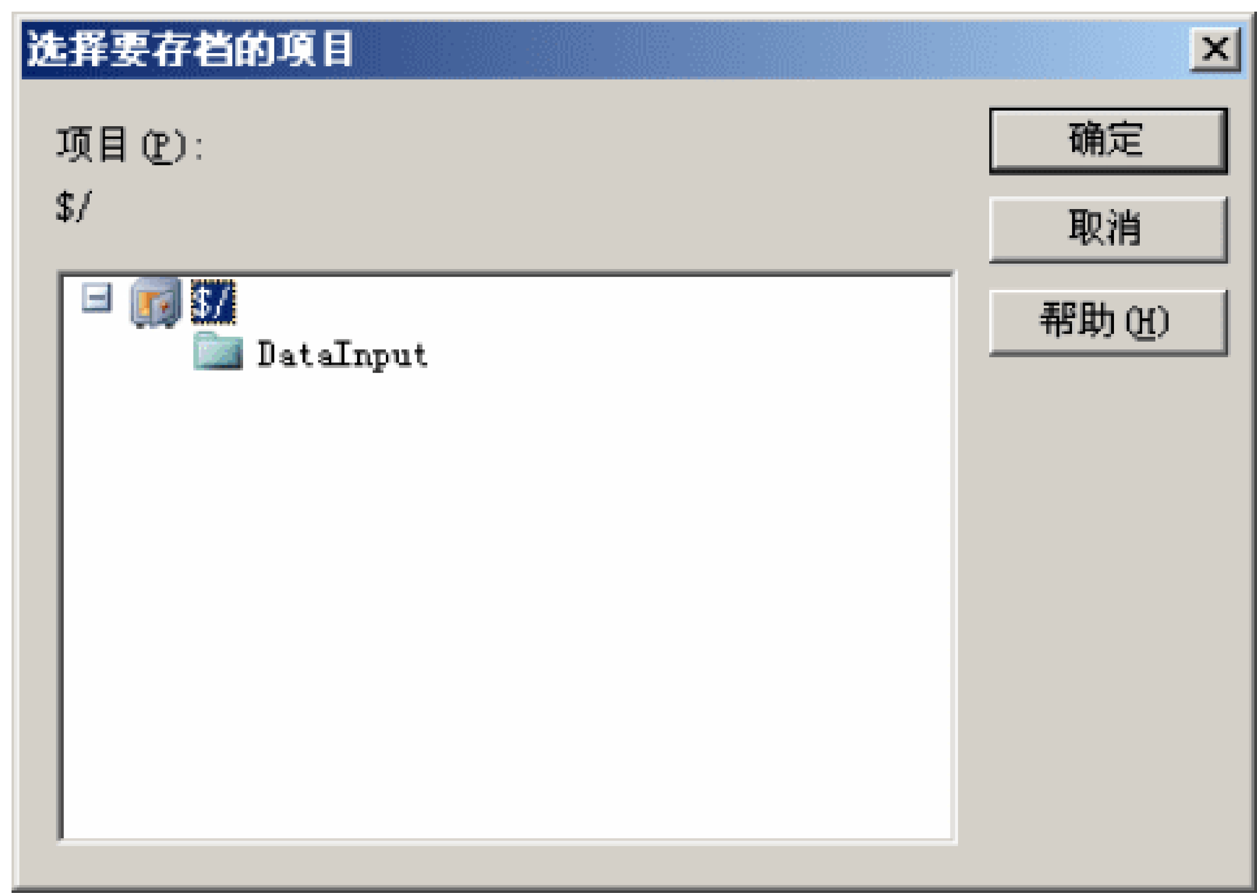


图 5.6　选择要存档的项目



图 5.7　存档向导

（5）在图 5.7 所示的存档向导对话框中单击“下一步”按钮，则进行如图 5.8 所示的选择存档方式对话框。有 3 个选项可供选择：将数据库备份到文件、备份并从数据库中删除、仅从数据中删除。根据实际情况选择一个选项，然后选择备份文件的保存路径，单击“下一步”按钮，直到完成为止。

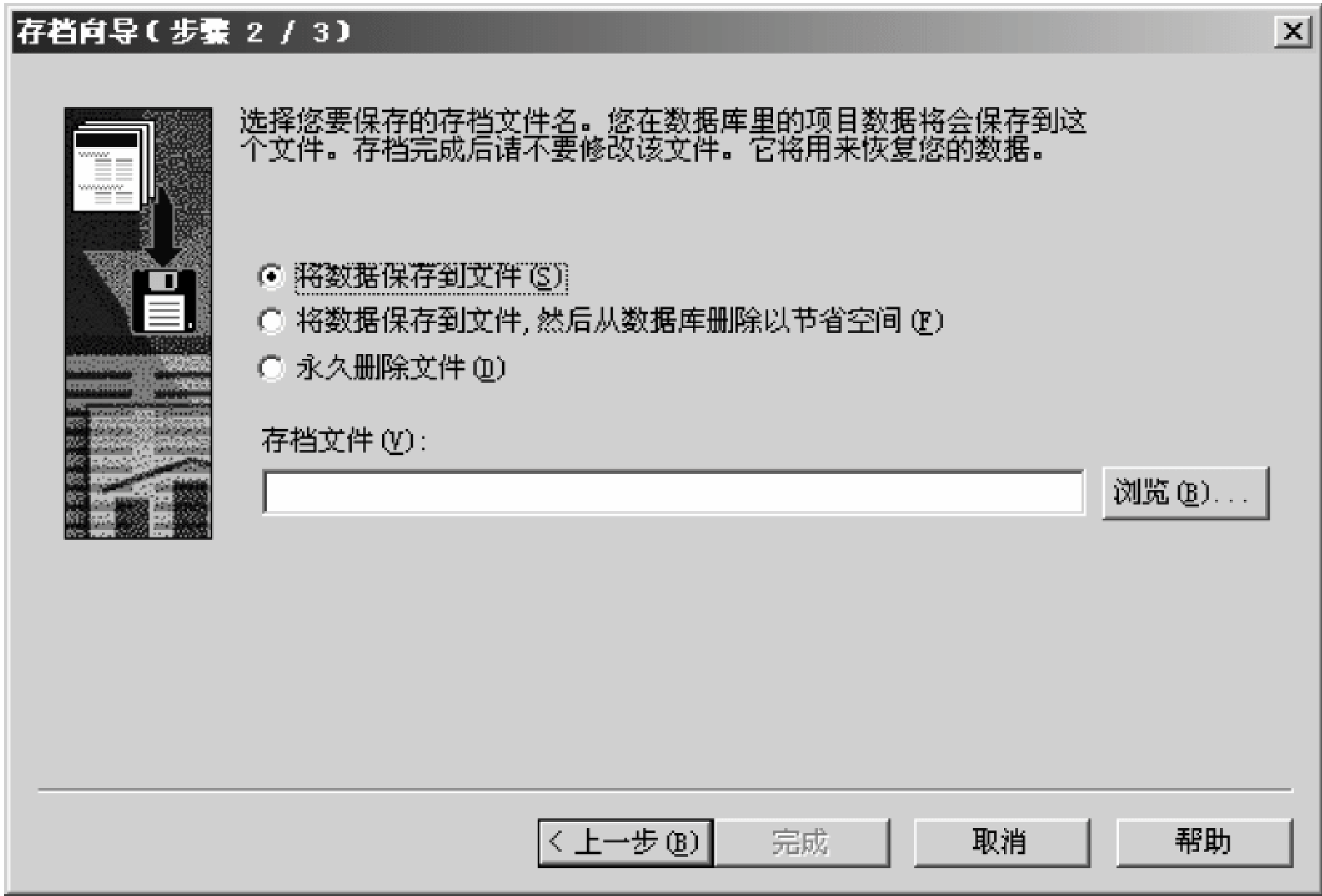


图 5.8　选择存档方式

（6）当 VSS 数据库备份完成后，会在指定的路径下生成一个文件，使用此文件可以恢复 VSS 数据库。如果要恢复 VSS 数据库，从 VSS 管理程序菜单中选择“存档”|“恢复项目”命令，则弹出恢复向导，在向导中选择要恢复的文件，一直单击“下一步”按钮直到数据库恢复完成。

5.2.3　配置 VSS 网络服务

当 VSS 用于团队开发时，团队成员需要从局域网基于 Internet 上访问 VSS 数据库，使用 VSS 所提供的签入、签出、版本比较等功能。在这种情况下，需要启用 VSS 的网络服务功能。操作步骤是从 VSS 管理程序的菜单中选择“服务器”|“配置”命令，则弹出如图 5.9 所示的“服务器配置”对话框。

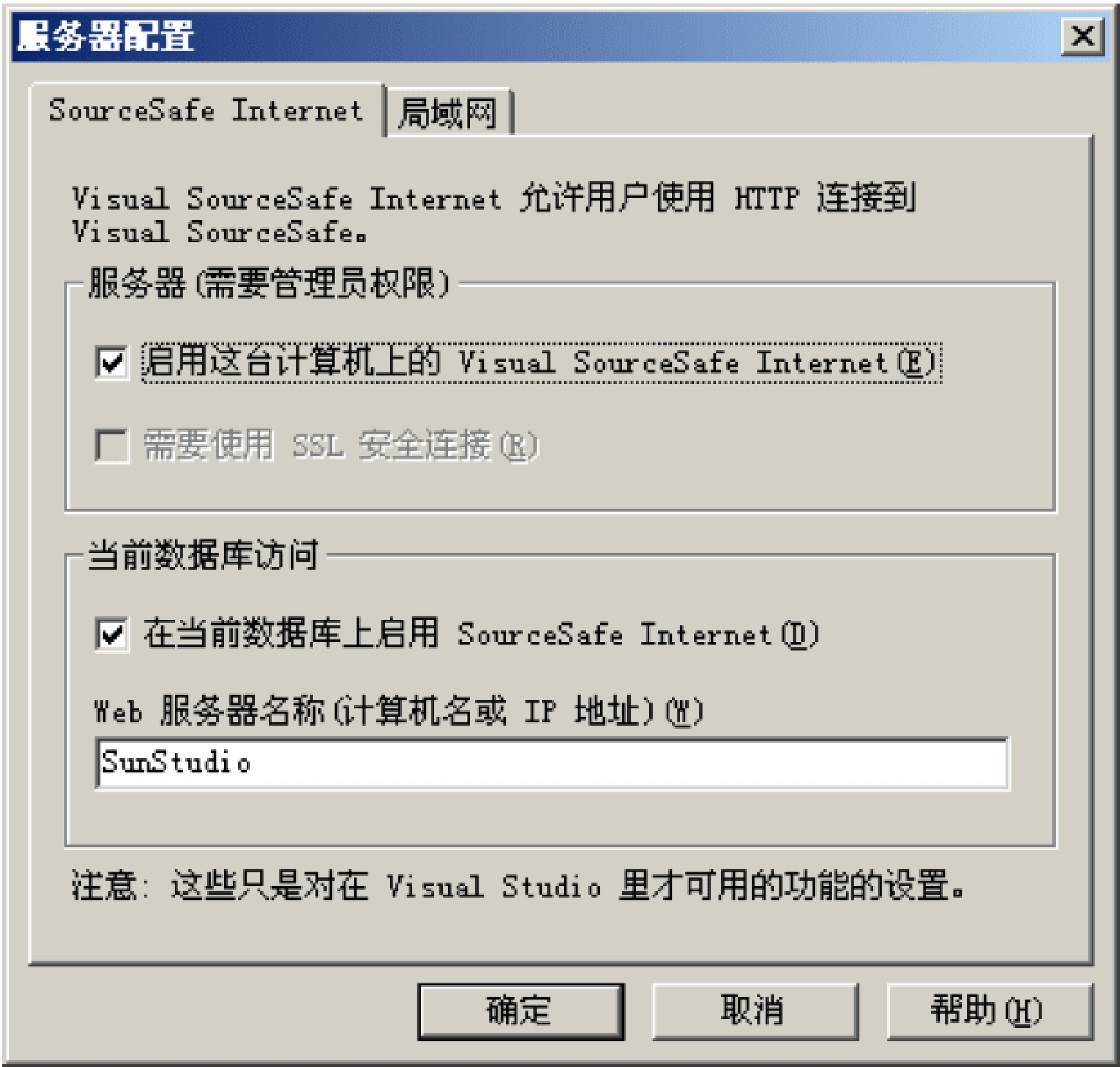


图 5.9　VSS 服务器配置对话框

如果要在 Internet 上使用 VSS，则在 SourceSafe Internet 选项卡中选中相应的选项，单击“确定”按钮。如果要在局域网使用 VSS，则在“局域网”标签页中进行相应的设置。设置成功后，即可通过 Internet 或者局域网访问此服务器上的 VSS 了。

提示： 启用网络服务的 VSS 数据库必须使用网络路径（如\\MyServer\VSSDB）而不能使用本地路径（如 D:\VSSDB），同时 VSS 数据库所在目录必须设置为网络共享且为 VSS 用户分配适当的读写权限。

5.2.4　VSS 源码管理

VSS 以项目为单位对源码进行管理，一个项目可以包含若干文件夹和文件。VSS 能够跟踪项目中文件的签入签出状态、历史版本、已删除文件等信息。

（1）要在 VSS 中添加一个项目，是从 VSS 程序菜单中选择“文件”|“创建项目”命令，在弹出的“创建项目”对话框中输入项目名称，单击“确定”按钮即可。项目添加以后，通常需要为其设置工作目录，VSS 的文件签出时就存放在工作目录中。右击 VSS 程序中一个项

目，从弹出菜单中选择“设置工作目录”命令，即可为 VSS 项目指定工作目录。

（2）在 VSS 中创建好项目并设置好工作目录以后，接下来要做的是把需要管理的文件添加到项目中。在 VSS 程序中，选择一个项目并右击，从弹出的快捷菜单中选择“添加文件”选项，弹出“添加文件”对话框。在其中选择文件并单击“打开”按钮。图 5.10 所示为添加了项目和文件以后的 VSS 程序主界面。

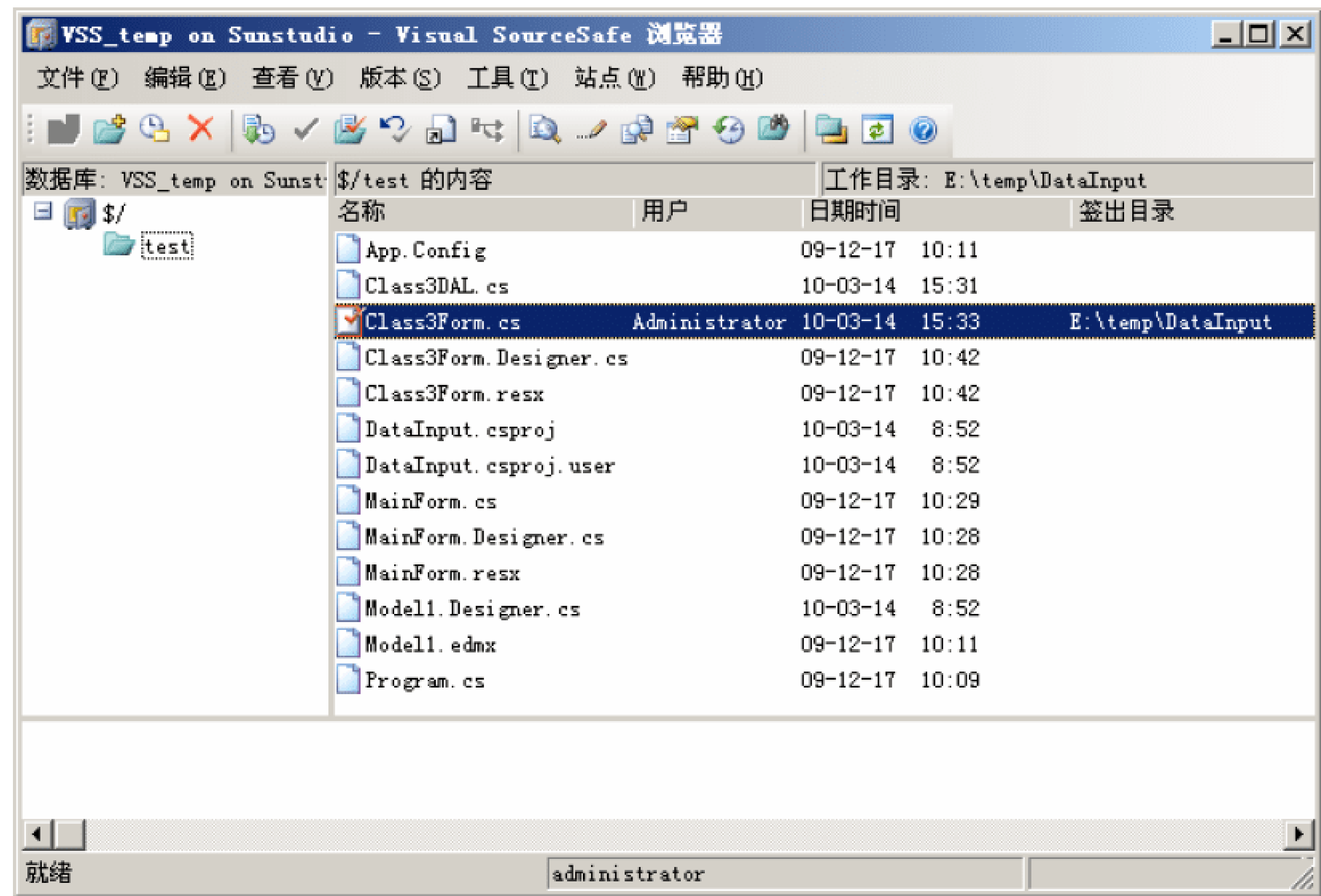


图 5.10　VSS 程序主界面

（3）在图 5.10 所示的 VSS 程序主界面中，列出了 test 项目下的所有文件，并用不同的图标表示文件的签入、签出状态。如果文件被签出，还显示签出文件的时间、目录和用户。要想签出并编辑文件，在 VSS 程序中选中要编辑的文件并右击，则弹出如图 5.11 所示的菜单，从菜单中选择“编辑”选项，则弹出如图 5.12 所示的编辑对话框。

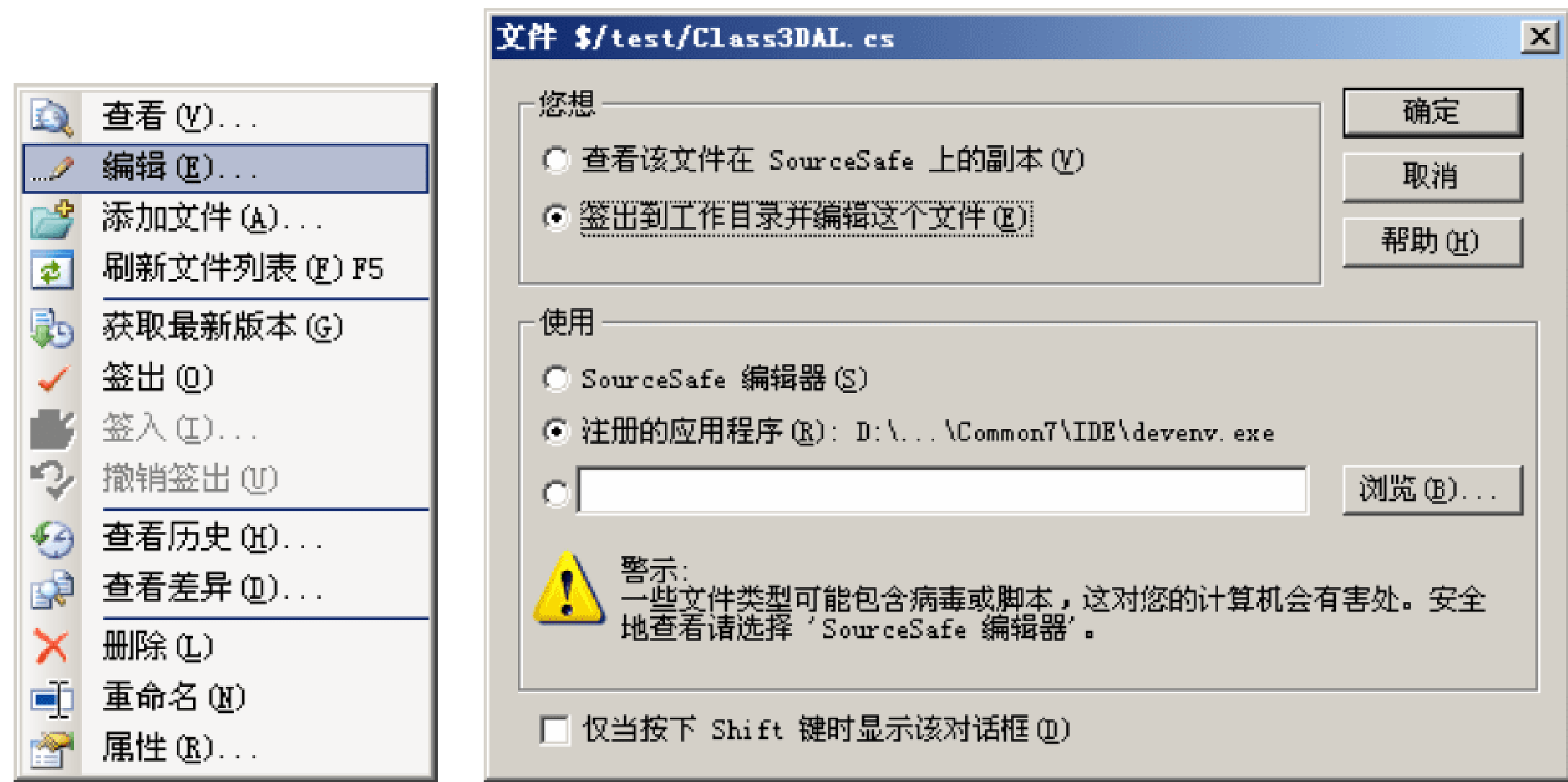


图 5.11　VSS 文件弹出菜单　　　　　　图 5.12　VSS 文件编辑对话框

（4）在图 5.12 所示的对话框中，可以指定要打开的文件所使用的编辑器，单击“确定”

按钮后，则将使用指定的编辑器打开文件。在编辑器中修改文件结束后，从 VSS 程序中单击文件并从弹出菜单中选择“签入”选项，即将修改后的文件签入到 VSS 服务器。VSS 中的文件每签入一次，就会产生一个新的版本。VSS 使用递增的整数自动为各个版本命名。在图 5.11 所示的文件弹出菜单中选择“查看历史”选项，则弹出如图 5.13 所示的文件历史对话框，在其中显示此文件的所有历史版本。

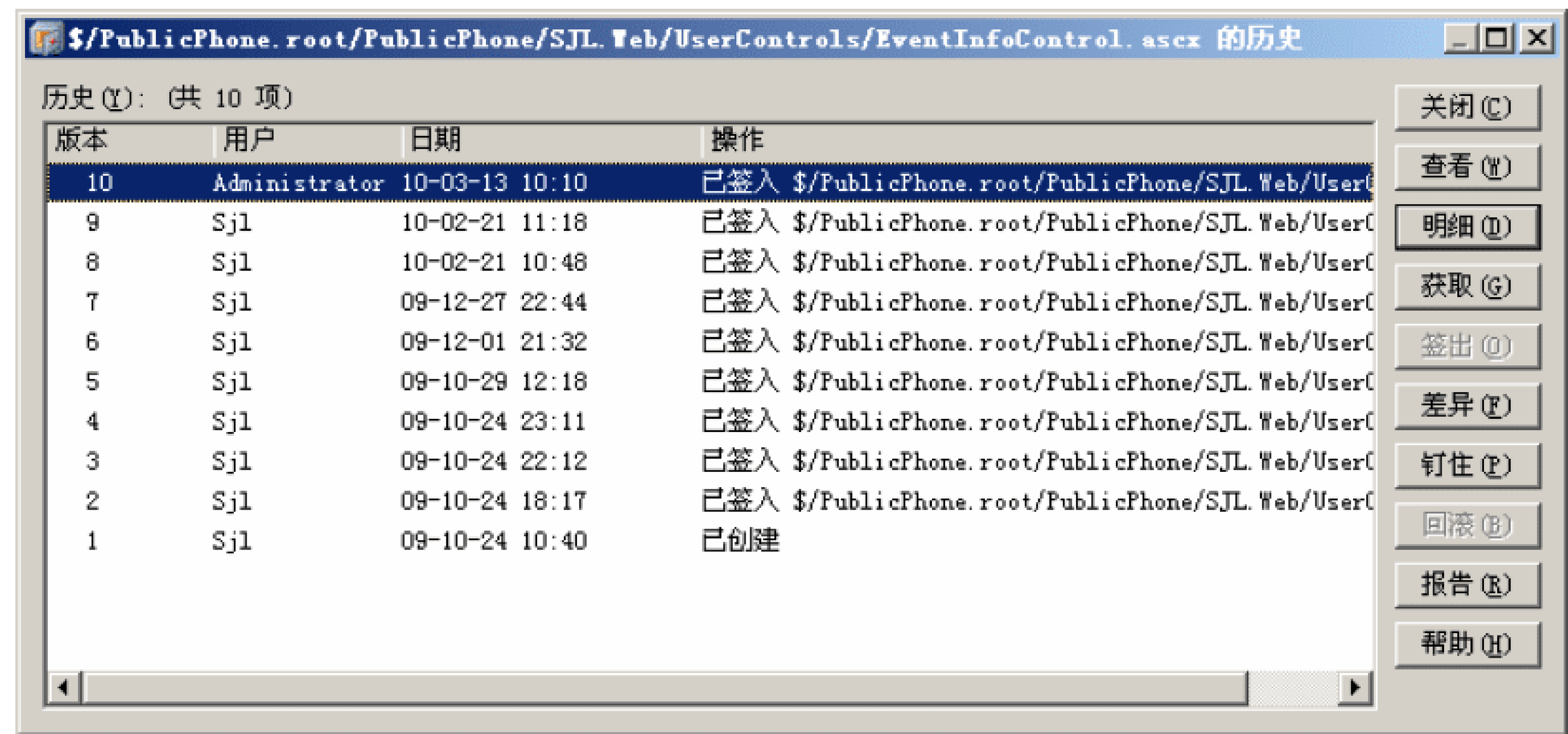


图 5.13　文件历史

（5）在图 5.13 的文件历史对话框中右侧有多个按钮，可以完成各种操作。选中一个版本，单击“查看”按钮，可以显示此版本的文件内容。单击“回滚”按钮，则可以将文件恢复到指定的版本。单击“获取”按钮，可以将选中版本导出为一个文件。在文件历史中选中两个不同版本，单击“差异”按钮，则可以查看两个版本文件的不同之处，如图 5.14 所示。VSS 用不同的颜色显示不同的文件差异，红色表示被删除的文本，蓝色表示发生改变的文本，绿色表示添加的文本，黑色表示未发生变化的文本。

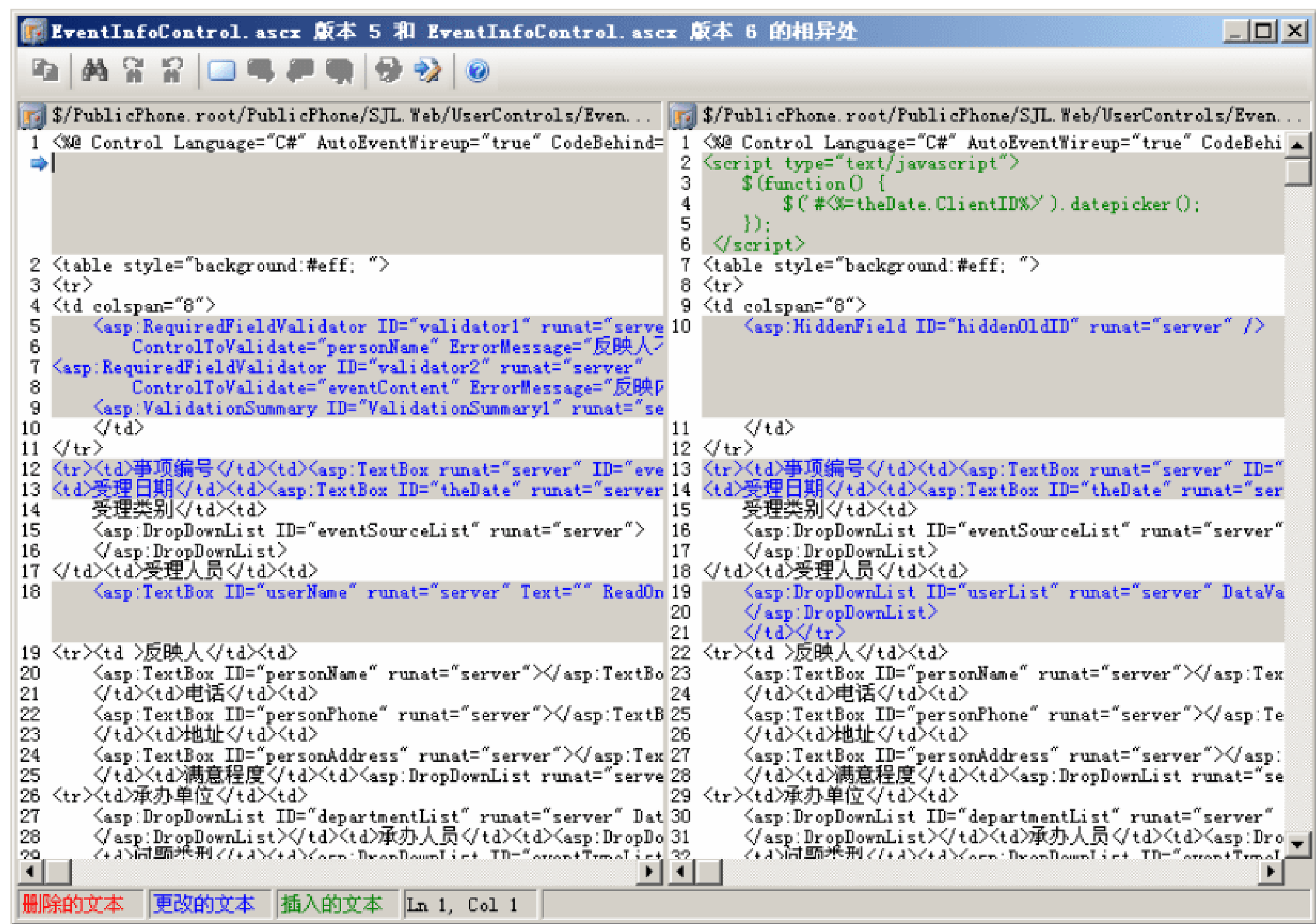


图 5.14　VSS 显示不同版本的文件差异

5.2.5　集成 Visual Studio 与 Visual SourceSafe

Visual Studio 开发环境和 Visual SourceSafe 都是微软的产品，二者能够协调地配合工作。在 Visual Studio 中，集成了 Visual SourceSafe 的客户端功能，可以很方便地实现签入、签出、查看历史、版本比较等功能。

提示： 默认情况下 Visual Studio 2010 使用 Team Foundation Server 实现源码管理功能。要想在 Visual Studio 2010 中使用 Visual SourceSafe 2005，需要安装一个插件，下载地址为 http://code.msdn.microsoft.com/KB976375。

Visual Studio 2010 支持包括 TFS 和 VSS 在内的多个源码管理软件，可以在 Visual Studio 2010 中进行配置，以指定一种源码管理工具。要使用 VSS 2005 作为默认源码管理工具，操作步骤为从菜单中选择“工具”|“选项”命令，在打开的选项对话框中，从左侧的树型列表中选择“源代码管理”|“插件选择”，并从下拉表框中选择 Microsoft Visual SourceSafe（如果要连接到 Internet 上的 VSS，则选择带 Internet 的 Visual SourceSafe），单击 OK 按钮，如图 5.15 所示。

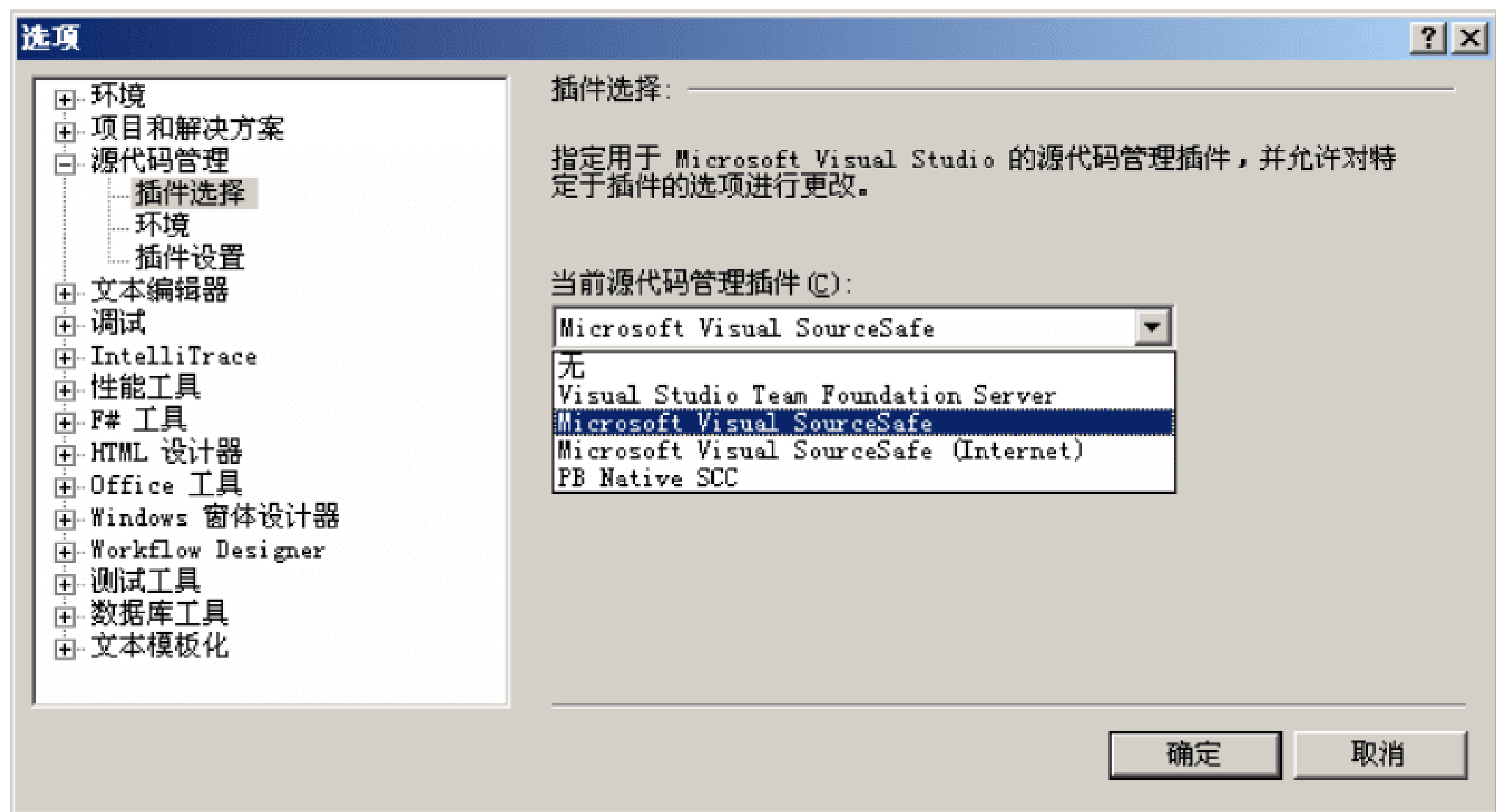


图 5.15　选择源码管理插件

如果要把 Visual Studio 2010 中的项目（此处以第 4 章所开发的 BookShop 项目为例）添加到 VSS 源码管理，则在解决方案资源管理器中右击项目名称，从弹出的快捷菜单中选择“将解决方案添加到源代码管理”选项，则会弹出 VSS 登录对话框，如图 5.16 所示。

在图 5.16 所示的 VSS 登录对话框中，选择正确的 VSS 数据库，输入合法用户名和密码，单击“确定”按钮，则会弹出如图 5.17 所示的“添加到 SourceSafe”对话框，让用户选择项目在 VSS 中的存储位置，一般使用默认值即可。

在图 5.17 所示对话框中选择一个位置并单击“确定”按钮后，整个项目中的文件就被加入到了 VSS 中。此时打开 VSS 程序，就可以看到这个项目，如图 5.18 所示。



图 5.16　VSS 登录对话框

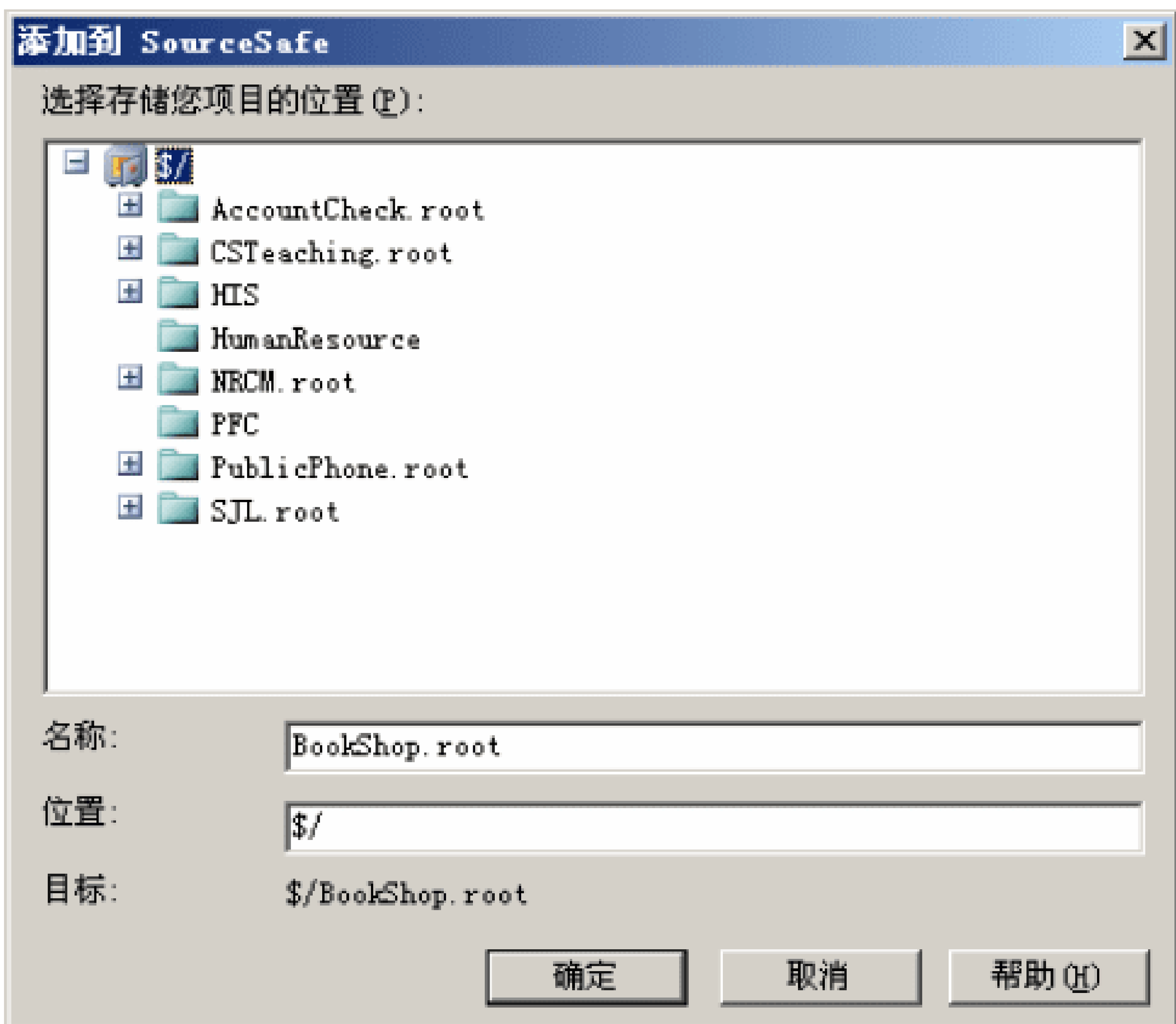


图 5.17　选择项目在 VSS 中的存储位置

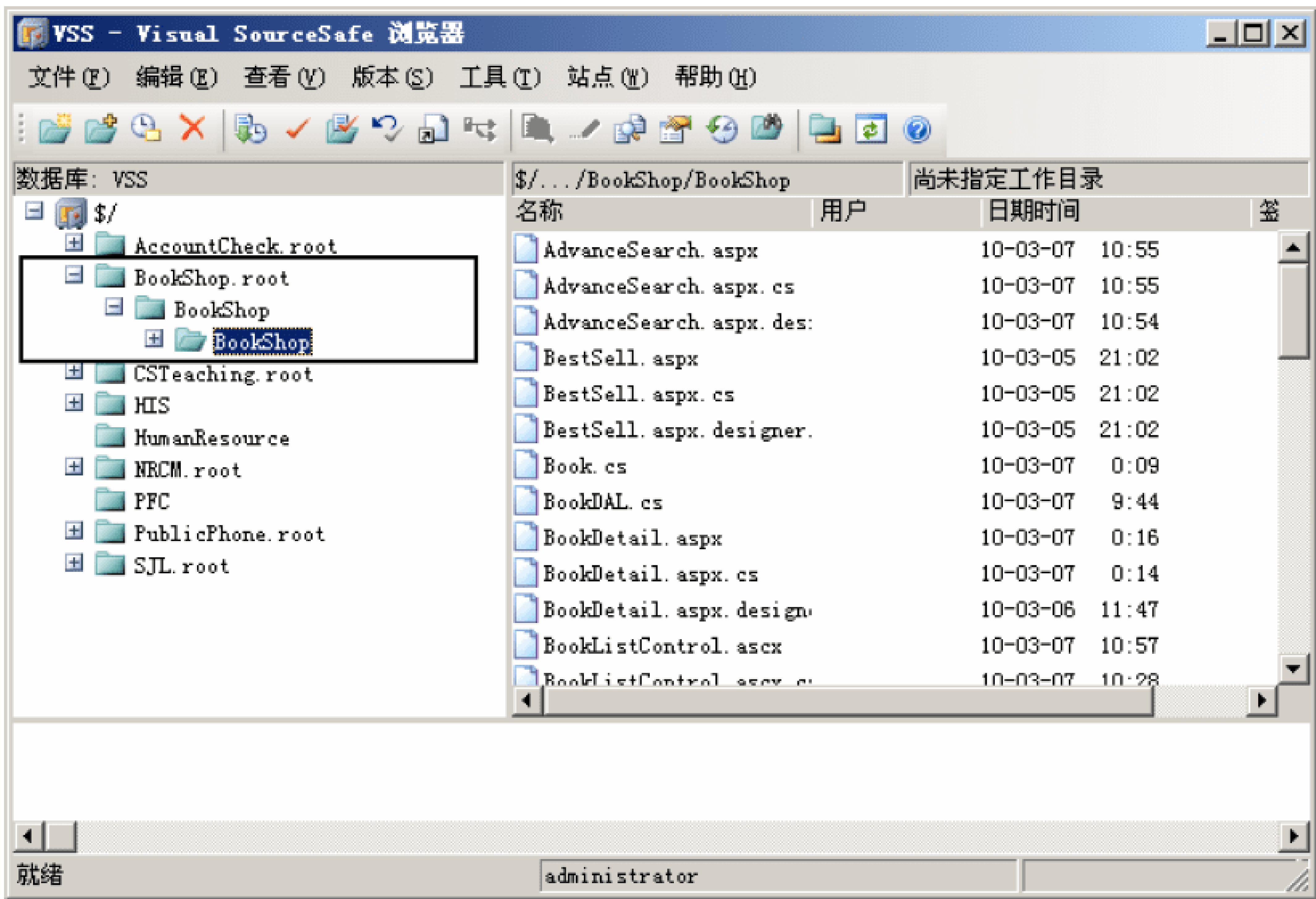


图 5.18　VSS 中新添加的 BookShop 项目

在 Visual Studio 中将项目添加到 Visual SourceSafe 源码管理以后，在解决方案资源管理器中文件的图标发生了变化，左侧添加了一把小锁（Default.aspx），表示此文件已经被签入 VSS 服务器。如果在 Visual Studio 中编辑这种被签入的文件时，文件会自动被签出，文件图标左侧的小锁图案变成对号（Default.aspx），表示此文件被当前开发人员签出。如果文件被其他开发人员签出，则文件图标左侧会出现一个小人头像（Default.aspx）。

在 Visual Studio 中进行源码文件的签入、查看历史、文件比较、获取版本等操作都可以通过文件右键菜单完成，具体操作方式与单独使用 VSS 时相同。

5.3　三 层 结 构

随着软件开发技术和理论的发展，软件体系架构也在不断发生着演化，从单机程序，

到客户端/服务器（C/S 结构）程序，到浏览器/服务器（B/S 结构）程序，到多层结构（Multi-tier Architecture）程序，到面向服务（SOA）程序，到软件即服务（SaaS）程序，到云计算（Cloud Computing）。软件体系结构的发展经历了多个的阶段，演化出不同的形式，在开发一个具体软件项目时，应该采用何种体系结构，是一个关系全局的重要问题。本节将介绍当前在 Web 开发中广为应用的三层结构。

5.3.1　三层结构概述

当编程实现一个功能时，开发人员最直观的思路是把所有代码都写到一起。举例来说，如果用 ASP.NET 实现一个用户登录页面时，开发人员会添加一个 ASP.NET 页面，上面放置相应的控件，然后在页面的后台代码中编写事件处理程序，将用户输入的用户名和密码与数据库中的数据进行验证，根据正确与否进行适当处理。这种思路虽然简单直观，但是仔细分析却存在很大的缺陷：功能混乱、不能复用、难以测试。

软件设计和开发人员经过长期的实践、思索和研究，提出了软件分层的概念，即把整个软件的功能分成多个层次，并形成了软件的多层架构。在软件的多层结构中，最为典型的是三层结构，即把软件分成表现层、业务逻辑层、数据访问层，各层之间通过业务实体类进行数据交换。典型的三层结构如图 5.19 所示。

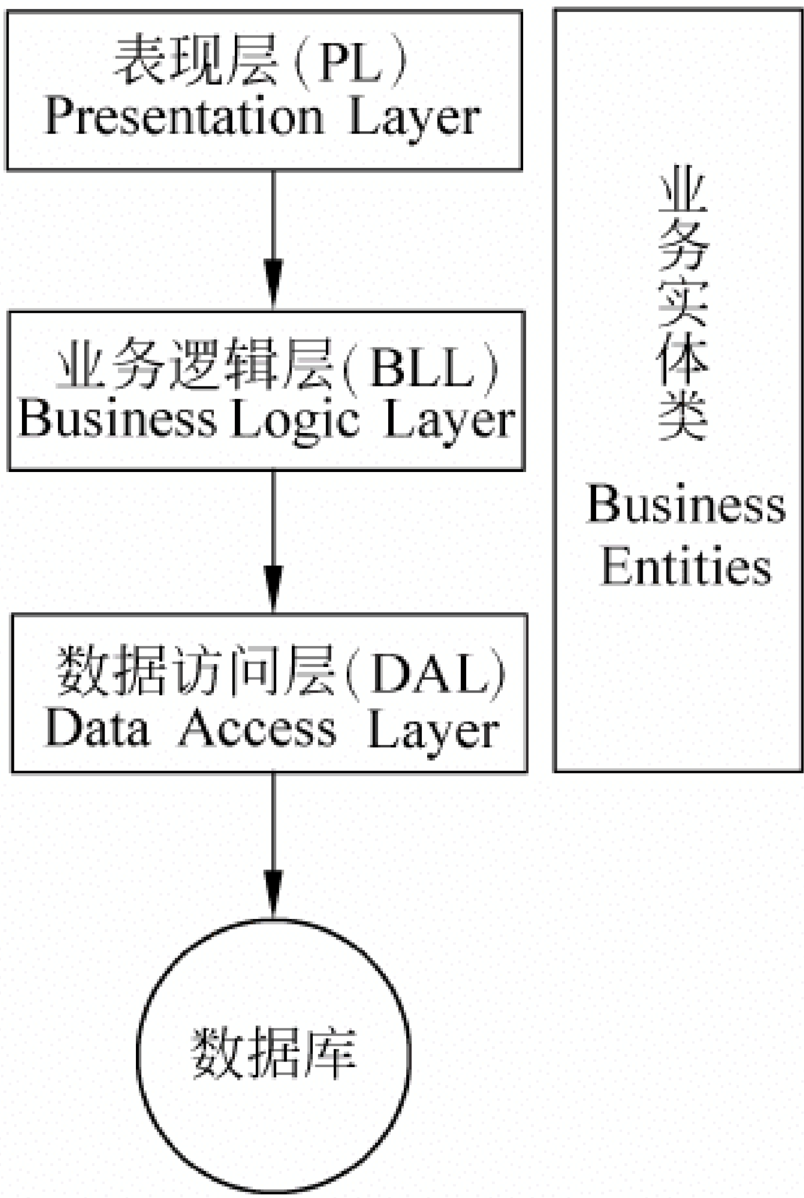


图 5.19　三层结构

在图 5.19 所示的三层结构中，表现层为用户界面，对于 ASP.NET 程序来说，表现层就是页面。业务逻辑层处理业务规则，例如，某系统规定用户登录时连续输错密码不得超过 5 次，否则用户账户将被锁定，这就是一个业务规则。数据访问层负责与数据库进行数据交换，执行各种数据库命令。

图 5.19 中的箭头表示调用和依赖关系。表现层、业务逻辑层、数据访问层各层之间是单向的逐层调用关系，不能跨层调用，也不能反向调用。例如，表现层只能调用业务逻辑层的功能，而不能直接调用数据访问层的功能，更不能直接操作数据库。数据访问层只能对数据库进行操作，而不能调用业务逻辑层的功能。

5.3.2　银行转账实例

为了更加形象地讲解三层结构的设计与实现，本节将引入一个小例子：银行转账程序，此程序可以将资金从一个银行账号转换到另一个银行账号。为了突出重点，本例将银行转账过程中与三层结构无关的功能进行了省略，对业务规则进行了加强。经过如此修改后的银行转账程序功能描述如下。

银行系统中每个客户都拥有一个用户名和支付密码，一个客户可以拥有零到多个账号。各个银行账号之间可以相互转账。转账时资金流出的账号称为源账号，资金注入的账

号称为目的账号。转账时需要输入以下信息：源账号、源账号客户名、源账号支付密码、转账金额、目的账号、目的账号客户名。程序需要对输入的信息进行以下验证，如果全部通过才能转账成功。

（1）源账号、客户名、支付密码都正确。

（2）源账号有足够的资金余额。

（3）源账号和目的账号都没有被冻结。

（4）转账金额不能超过某一上限，具体数值由银行指定。

（5）目的账号用户名和密码都正确。

（6）源账号当天累计转出资金不超过某一上限，具体数值由银行指定。

（7）源账号每天进行转出交易笔数不超过某一上限，具体数值由银行指定。

银行系统数据库包括以下 4 个表。

（1）顾客信息表 Customer，存储银行顾客信息，表中各字段如下。

- ❑ CustomerID：顾客编号，nvarchar(20)类型，主键。
- ❑ CustomerName：顾客名称，nvarchar(10)类型，不可为空。
- ❑ LoginPassword：登录密码，nvarchar(20)类型，不可为空。
- ❑ PayPassword：支付密码，nvarchar(20)类型，不可为空。

（2）银行账户表 BankAccount，存储银行账号信息，表中各字段如下。

- ❑ AccountNo：银行账号，nvarchar(20)类型，主键。
- ❑ CustomerID：顾客编号，nvarchar(20)类型，外键关联到 Customer 表，不可为空。
- ❑ Balance：账户余额，float 类型，不可为空。
- ❑ IsLocked：是否被冻结，bit 类型，不可为空，默认值为 false。

（3）账户交易表 AccountTransaction，存储账号交易明细，表中各字段如下。

- ❑ TransactionID：交易编号，int 类型，自动标识列，主键。
- ❑ TransactionDate：交易日期，datetime 类型，默认为当前时间，不可为空。
- ❑ SourceAccount：源账号，nvarchar(20)类型，外键关联到 BankAccount 表，不可为空。
- ❑ DestinationAccount：目的账号，nvarchar(20)类型，外键关联到 BankAccount 表，不可为空。
- ❑ TransferMoney：转账金额，float 类型，不可为空。

（4）银行政策表 BankPolicy，存储银行政策（如单笔转账金额上限、每日转账交易上限等），表中各字段如下。

- ❑ PolicyID：政策 ID，nvarchar(10)类型，主键。
- ❑ Description：政策描述，nvarchar(50)类型。
- ❑ PolicyContent：政策内容，如交易上限数值，nvarchar(50)类型。

5.3.3　未分层的银行转账程序

通过比较多层体系结构与不分层的软件之间的区别，能够更好地体会多层结构的优势。本节先采用传统的不分层的方式实现上述银行转账程序，并分析这种方案的缺陷，然

后在下一节采用三层结构对银行转账程序进行重构。

（1）创建一个 ASP.NET Web 应用程序。

（2）在 SQL Server 中添加一个数据库，按照 5.3.2 节所设计的数据库结构在数据库中添加各个表。在 web.config 文件中添加对此数据库的连接字符串。

（3）在项目中添加一个页面 SimpleBankPage.aspx，在页面上放置多个 TextBox 供用户输入账号、密码、金额等数据。页面代码如下：

```
<form id="form1" runat="server">
<div>
<h3>银行转账程序</h3>
<table><tr>
<td>客户编号：<asp:TextBox ID="customer" runat="server"></asp:TextBox></td>
<td>登录密码：
<asp:TextBox ID="loginPass" runat="server" TextMode="Password">
</asp:TextBox></td>
<td>支付密码：
<asp:TextBox ID="payPass" runat="server" TextMode="Password">
</asp:TextBox></td>
</tr><tr>
<td>转出账号：<asp:TextBox ID="source" runat="server"></asp:TextBox></td>
 <td>转账金额：<asp:TextBox ID="money" runat="server"></asp:TextBox></td>
<td></td></tr>
<tr>
<td> 接收账号：<asp:TextBox ID="destination" runat="server">
</asp:TextBox></td>
<td> 账号名称：<asp:TextBox ID="destinationName" runat="server">
</asp:TextBox></td>
<td><asp:Button ID="ok" runat="server" Text="转账" Width="100" onclick=
"ok_Click" />
</td></tr>
</table>
<asp:Label runat="server" ID="error" ForeColor="Red" Visible="false">
</asp:Label>
<asp:Label runat="server" ID="message" Visible="false" ></asp:Label>
</div>
</form>
```

（4）在 SimpleBankPage.aspx.cs 中编写一个方法，对顾客输入的客户 ID、登录密码和支付密码进行判断，并检测该用户是否拥有所输入的源账号。

```
/// <summary>
/// 验证顾客信息，客户 ID、登录密码、支付密码、是否拥有源账号
/// </summary>
/// <returns>验证合法返回 true，否则 false</returns>
private bool checkPassword()
{
    using (SqlHelper db = new SqlHelper())
    {
        //从数据库读取顾客编号和两个密码（登录密码、支付密码）
        string sql = "select LoginPassword,PayPassword from Customer where
        CustomerID=@id";
        using (DbCommand command = db.GetSqlStringCommond(sql))
        {
            db.AddInParameter(command, "@id", DbType.String, customer. Text
            );
            using (DbDataReader reader = db.ExecuteReader(command))
```

```
            {
                //如果不存在此顾客 ID，则显示错误提示，返回 false
                if (!reader.Read())
                {
                    showError("不存在你所输入的顾客 ID。");
                    return false;
                }
                string pass1, pass2;
                pass1 = reader[0].ToString();
                pass2 = reader[1].ToString();
                //比较两个密码，如果不正确，则显示错误提示，返回 false
                if (loginPass.Text != pass1 || payPass.Text != pass2)
                {
                    showError("密码不正确。");
                    return false;
                }
            }
        }
        //检测用户是否拥有所输入的源账号
        sql = "select count(*) from BankAccount where AccountNo=@no and
        CustomerID=@id";
        using (DbCommand command = db.GetSqlStringCommond(sql))
        {
            db.AddInParameter(command, "@id", DbType.String, customer. Text
            );
            db.AddInParameter(command, "@no", DbType.String, source.Text);
            int n = Convert.ToInt32(db.ExecuteScalar(command));
            if (n == 0)
            {
                showError("未找到你所输入的账号。");
                return false;
            }
        }
    }
    return true;                                    //所有检测合法，返回 true
}
```

（5）在 SimpleBankPage.aspx.cs 中添加一个方法，对源账号信息进行验证，包括检验源账号是否被冻结，以及源账号中是否有足够的余额以供转账。

```
//检查源账号是否被冻结以及资金余额是否充足
private bool checkSourceAccount()
{
    using (SqlHelper db = new SqlHelper())
    {
        //从数据库获取账户状态（是否冻结，资金余额）
        string sql = "select IsLocked,Balance from BankAccount where
        AccountNo=@no";
        using (DbCommand command = db.GetSqlStringCommond(sql))
        {
            db.AddInParameter(command, "@no", DbType.String, source.Text);
            using (DbDataReader reader = db.ExecuteReader(command))
            {
                if (!reader.Read())                 //账号是否存在
                {
                    showError("你输入的账号["+source.Text+"]不存在。");
                    return false;
                }
                bool locked = Convert.ToBoolean(reader[0]);       //是否冻结
```

```
                double balance = Convert.ToDouble(reader[1]);    //账号余额
                reader.Close();
                if (locked)
                {
                    showError("源账号["+source.Text+"]已经被冻结。");
                    return false;
                }
                if (balance < double.Parse(money.Text))
                {
                    showError("源账号["+source.Text+"]资金余额不足。");
                    return false;
                }
            }
        }
        return true;
    }
}
```

（6）在 SimpleBankPage.aspx.cs 中添加一个方法，检验所转账金额是否超出数据库中定义的最大单笔转账金额，检验源账号当天转账次数是否已经达到最大转账次数，检验此次转账是否会使源账号的转账总额超出最大当日转账金额。

```
/// <summary>
/// 检查交易上限（包括单笔金额上限，累计转账金额上限，累计交易次数上限）
/// </summary>
/// <returns>检查是否合法</returns>
private bool checkTransactoinLimit()
{
    using (SqlHelper db = new SqlHelper())
    {
        double singleTranserLimit;                        //单笔转账金额上限
        int transferTimesLimit;                           //单日累计转账次数上限
        double transferMoneyLimit;                        //单日累计转账金额上限
         //从数据库读取银行政策（3 个上限值）
        string sql = "select PolicyContent from BankPolicy where PolicyID
        =@id";
        using (DbCommand command = db.GetSqlStringCommond(sql))
        {
            //从数据库读取单笔转账金额上限
            db.AddInParameter(command, "@id", DbType.String, "policy001");
            singleTranserLimit = Convert.ToDouble(db.ExecuteScalar
            (command));
            //从数据库读取单日累计转账次数上限
            command.Parameters["@id"].Value = "policy002";
            transferTimesLimit = Convert.ToInt32(db.ExecuteScalar
            (command));
            //从数据库读取单日累计转账金额上限
            command.Parameters["@id"].Value = "policy003";
            transferMoneyLimit = Convert.ToDouble(db.ExecuteScalar
            (command));
        }
        //检验单笔转账上限
        if (singleTranserLimit < double.Parse(money.Text))
        {
            showError("你要转账的金额超过单笔转账上限"+singleTranserLimit.
            ToString());
            return false;
        }
```

```
        //从数据库读取源账号当日累计转账次数和当时累计转账金额
        sql = "select count(*),sum(TransferMoney) from AccountTransaction "
            +" where SourceAccount=@no";
        using (DbCommand command = db.GetSqlStringCommond(sql))
        {
            db.AddInParameter(command, "@no", DbType.String, source.Text);
            using (DbDataReader reader = db.ExecuteReader(command))
            {
                if (reader.Read())
                {
                    int times = Convert.ToInt32(reader[0]);
                    double theMoney=0;
                    if(reader[1]!=DBNull.Value)
                        theMoney= Convert.ToDouble(reader[1]);
                    if (times == transferTimesLimit)          //检查累计转账次数
                    {
                        showError("账号["+source.Text
                              +"]当日累计转账次数已经达到上限，不能进行转账。");
                        return false;
                    }
                    //检查累计转账金额
                    if (theMoney+double.Parse(money.Text) > transferMoneyLim-
                    it)
                    {
                        showError("此次转账将使账号["
                             + source.Text + "]当日累计转账金额超出上限，不能进行转
                             账。");
                        return false;
                    }
                }
            }                                              //using DbDataReader
        }                                                  //using DbCommand
    }                                                      //using SqlHelper
    return true;
}
```

（7）在 SimpleBankPage.aspx.cs 中添加一个方法，检测目的账号是否合法，包括目的账号是否存在，账号所对应的客户名称与输入的是否一致，目的账号是否已经被冻结。

```
/// <summary>
/// 检查目的账号（账号是否存在，名称是否正确，是否被冻结）
/// </summary>
/// <returns>检查是否合法</returns>
private bool checkDestinationAccount()
{
    using (SqlHelper db = new SqlHelper())
    {
        //从数据库获取账户状态（是否冻结，顾客 ID，顾客名称）
        string sql = "select IsLocked,CustomerID from BankAccount where
        AccountNo=@no";
        bool locked;
        string customerid, customerName;
        using (DbCommand command = db.GetSqlStringCommond(sql))
        {
            db.AddInParameter(command, "@no", DbType.String, destination.
            Text);
            using (DbDataReader reader = db.ExecuteReader(command))
            {
                if (!reader.Read())
```

```
                {
                    showError("你输入的账号["+destination.Text+"]不存在。");
                    return false;
                }
                locked = Convert.ToBoolean(reader[0]);
                customerid = Convert.ToString(reader[1]);
            }
        }
        sql = "select CustomerName from Customer where CustomerID=@id";
        using (DbCommand command = db.GetSqlStringCommond(sql))
        {
            db.AddInParameter(command, "@id", DbType.String, customerid);
            customerName = Convert.ToString(db.ExecuteScalar(command));
        }
        if (customerName != destinationName.Text)    //输入目的账号名称不对

        {
            showError("目的账号[" + source.Text + "]名称不对，请检查账号是否正确。
            ");
            return false;
        }
        if (locked)                                 //账号被冻结
        {
            showError("目的账号[" + source.Text + "]已经被冻结。");
            return false;
        }
    }
    return true;
}
```

（8）在 SimpleBankPage.aspx.cs 中添加一个方法，完成具体的转账操作，包括三部分内容：减少源账号金额、增加目的账号金额、记录交易详情。为了保证数据完整性，这三个操作应该在同一事务中完成。

```
//执行转账操作
private void transerMoney()
{
    using (SqlHelper db = new SqlHelper())
    {
        using (Trans tran = new Trans())                    //启动事务
        {
            //修改源账号余额
            string sql = "update BankAccount set Balance=Balance-@money where
            AccountNo=@no";
            using (DbCommand command = db.GetSqlStringCommond(sql))
            {
                db.AddInParameter(command,"@no", DbType.String,destination.
                Text);
                db.AddInParameter(command, "@money", DbType.Double, money.
                Text);
                db.ExecuteNonQuery(command, tran);
            }
            //修改目的账号余额
            sql = "update BankAccount set Balance=Balance+@money where
            AccountNo=@no";
            using (DbCommand command = db.GetSqlStringCommond(sql))
            {
                db.AddInParameter(command, "@no", DbType.String, source.
                Text);
```

```
                db.AddInParameter(command, "@money", DbType.Double, money.
                Text);
                db.ExecuteNonQuery(command, tran);
            }
            //将交易记录保存到数据库
            sql = "insert into AccountTransaction (SourceAccount, Destination
            Account, TransferMoney) "+ " values (@source,@dest,@money)";
            using (DbCommand command = db.GetSqlStringCommand(sql))
            {
                db.AddInParameter(command, "@source", DbType.String, source.
                Text);
                db.AddInParameter(command, "@dest", DbType.String, destina-
                tion.Text);
                db.AddInParameter(command, "@money", DbType.Double, money.
                Text);
                db.ExecuteNonQuery(command, tran);
            }
            tran.Commit();                                  //提交事务
        }
    }
}
```

（9）在“转账”按钮的 Click 事件中，调用以上方法，实现所有验证和转账操作。

```
protected void ok Click(object sender, EventArgs e)
{
    if (!checkPassword())
        return;
    if (!checkSourceAccount())
        return;
    if (!checkTransactoinLimit())
        return;
    if (!checkDestinationAccount())
        return;
    transerMoney();
    showMessage("转账操作成功。");
}
```

（10）在 SimpleBankPage.aspx.cs 中添加两个方法，分别用于显示错误信息和交易成功的提示信息。

```
/// <summary>
/// 显示错误提示，同时隐藏操作结果提示
/// </summary>
/// <param name="errorMessage">要显示的错误信息</param>
private void showError(string errorMessage)
{
    error.Text = errorMessage;
    error.Visible = true;
    message.Visible = false;
    message.Text = "";
}
/// <summary>
/// 显示操作提示（如操作成功），同时隐藏错误提示
/// </summary>
/// <param name="content">要显示的操作提示内容</param>
private void showMessage(string content)
{
    message.Visible = true;
```

```
    error.Visible = false;
    error.Text = "";
    message.Text = content;
}
```

（11）运行程序，运行界面如图 5.20 所示。

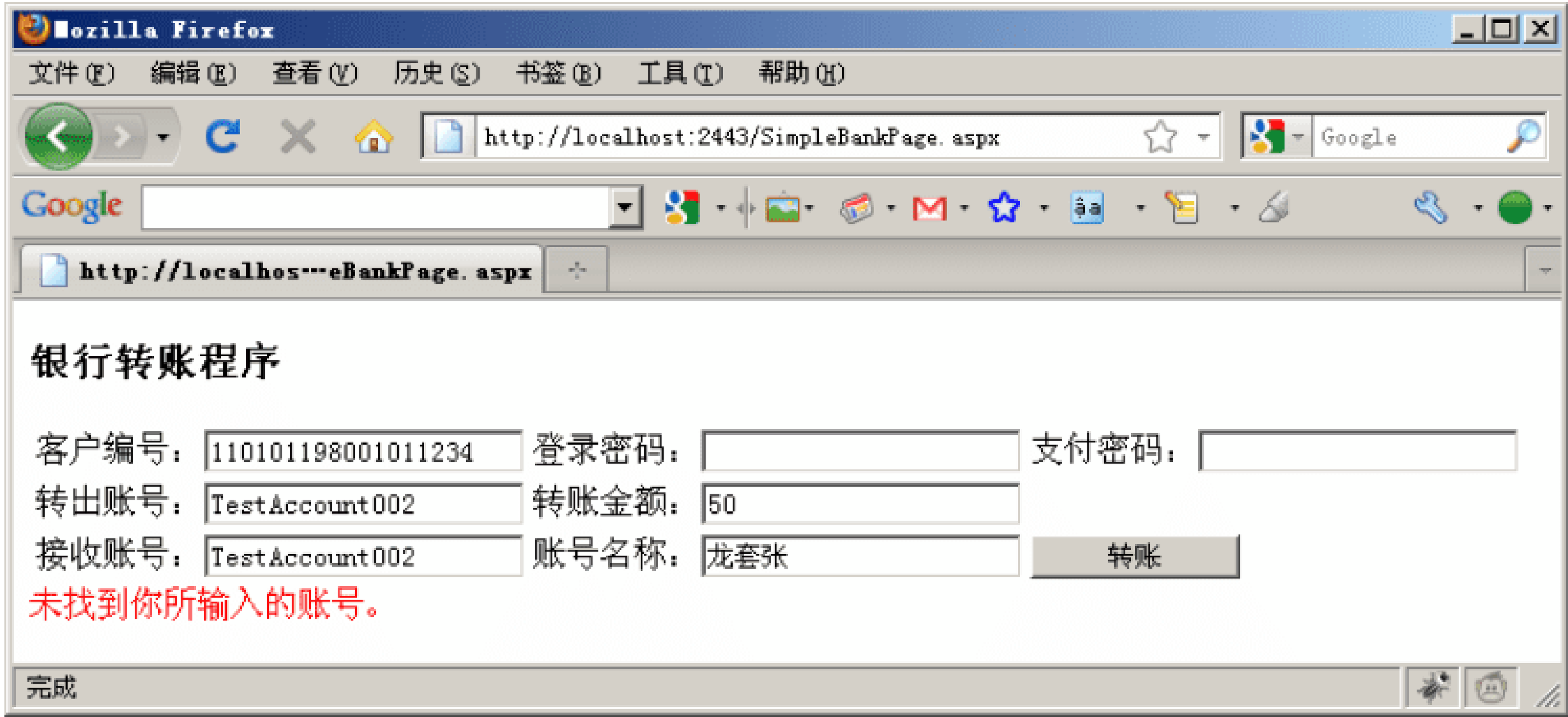


图 5.20　银行转账程序运行界面

5.3.4　未分层程序的缺陷

5.3.3 节的未分层的银行转账程序虽然实现了所要求的功能，但是存在以下缺陷。

（1）功能耦合。在银行转账页面中，要处理顾客信息、账号信息、转账交易、银行政策、用户界面等各方面的功能，不符合面向对象设计原则中的单一职责原则，是一种不好的设计。

（2）不可复用。如果做一个完整的网上银行系统，则在银行转账页面中包含的一些功能可能在其他多个页面中用到。例如，验证顾客密码功能在登录页面中会用到，查询账户余额功能在账户详情页面中会用到，查询账户是否锁定在账户的其他交易中会用到。由于这些代码与银行转账页面紧密耦合在一起，就无法在其他页面中调用，而是需要将这些代码复制到其他页面中，并且进行适当修改。这种做法违反了一个功能只实现一次的软件设计原则，如果要对某一功能做修改，则需要修改相似代码的多个复本，既增加了工作量，而且容易遗漏。

（3）不易修改。如果由于业务规则发生变化或者数据库结构发生变化而需要修改代码，由于代码分散在几乎所有的页面中，很难找到并改正所有需要修改的地方。

（4）文件庞大。由于功能复杂，银行转账页面代码数量太多，不易阅读和维护。

（5）弱类型。由于使用 DataReader（或者 DataSet）操作数据，得到的数据总是 object 类型，在具体使用时需要进行频繁的类型转换。这样一方面代码编写太复杂，另一方面当项目规模较大时，开发人员很难记得住每个字段的类型，容易引起类型转换错误。

5.3.5　三层结构的银行转账程序

正如前面所分析，当使用早期的不分层的软件结构开发较为复杂的系统时，暴露出种

种弊端。使用分层的软件体系结构来开发软件，能够克服这些弊端，设计出优秀的软件。在软件的多层结构中，最为经典的是三层结构，包括表现层、业务逻辑层和数据访问层。

当用.NET 开发应用程序时，通常每一层是一个单独的项目，其中表现层是 ASP.NET 项目（或者 Windows 应用程序项目），业务逻辑层和数据访问层都是类库项目。另外，还需要一个单独的类库项目，封装所有的业务实体类。用 ASP.NET 创建多层应用程序的第一个步骤就是在一个解决方案中创建以上 4 个项目。以实现银行转账程序为例，创建三层结构的项目步骤如下。

（1）创建一个 ASP.NET Web 应用程序 BankApp。

（2）在同一解决方案中添加一个类库项目 BankBLL，作为业务逻辑层。

（3）在同一解决方案中添加一个类库项目 BankDAL，作为数据访问层。

（4）在同一解决方案中添加一个类库项目 BankEntity，作为业务实体类库。

（5）在以上 4 个项目之间建立引用和依赖关系，BankBLL 引用 BankDAL，BankApp 引用 BankBLL，三个项目都引用 BankEntity。4 个项目之间的引用关系如图 5.21 所示。

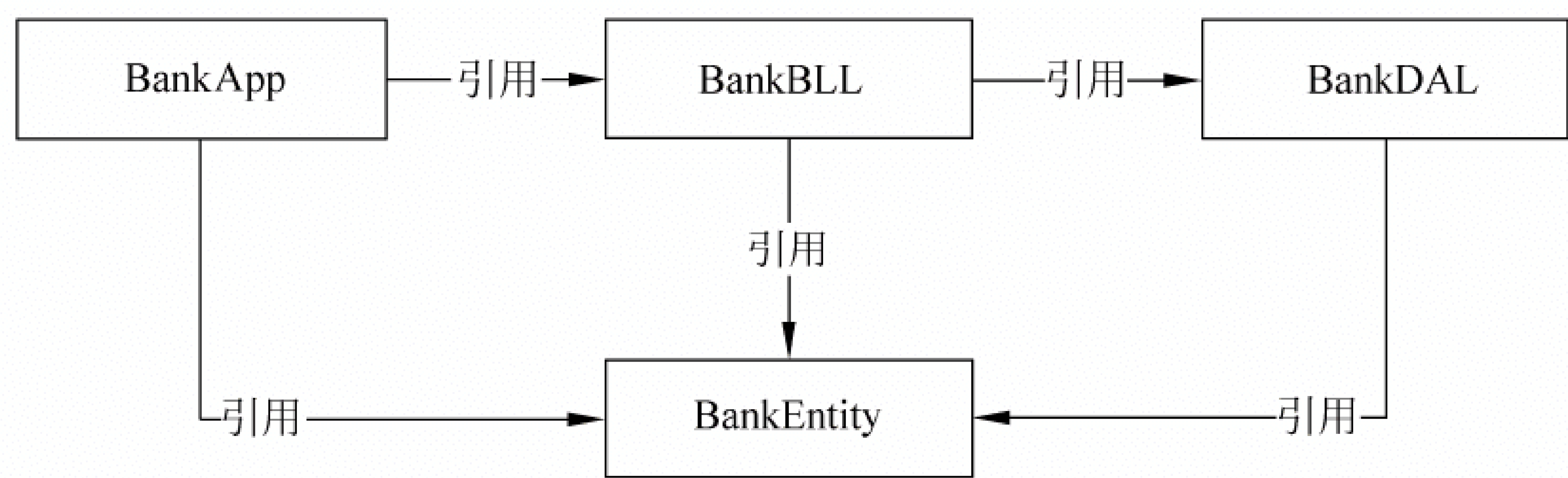


图 5.21　银行转账程序各项目之间引用关系

搭建好 4 个项目以后，接下来的工作就是将整个程序功能分配到各个层。表现层负责处理用户界面（如显示信息、接收用户输入），业务逻辑层实现业务逻辑（如本例中的银行业务规则），数据访问层负责操作数据库（执行增删改查），业务实体层包含项目中的业务实体类。在较为简单的情况下，数据库中的一个表对应于一个实体类，表中每个字段对应实体类的一个属性。

（6）在 BankEntity 中添加 4 个实体类，代码如下。

提示： 在 Visual Studio 中添加类时，默认情况下 class 关键字前面没有访问修饰符，此时类的访问级别为 internal，只能在当前程序集中访问。对于实体类来说，需要在其他项目访问，所以在 class 关键字前面添加 public 关键字，将类声明为公书共的。

```
//银行客户实体类
public class Customer
{
    public string id { get; set; }
    public string name { get; set; }
    public string loginPassword { get; set; }
    public string payPassword { get; set; }
}
//银行账号实体类
public class BankAccount
```

```
{
    public string no { get; set; }
    public double balance { get; set; }
    public string customerID { get; set; }
    public bool isLocked { get; set; }
}
//账号交易信息实体类
public class AccountTransaction
{
    public int transactionID { get; set; }
    public string sourceAccount { get; set; }
    public string destinationAccount { get; set; }
    public DateTime transactionDate { get; set; }
    public double transerMoney { get; set; }
}
//银行政策实体类
public class BankPolicy
{
    public string id { get; set; }
    public string description { get; set; }
    public string content { get; set; }
}
```

（7）在数据访问层 BankDAL 项目中添加一个 CustomerDal 类，完成与顾客信息相关的数据访问。本例中，对于顾客信息的数据访问只涉及一个功能，即根据顾客编号得到顾客详情，因此 CustomerDal 类中只有一个方法。

```
public  static class CustomerDal
{
    //根据顾客 ID 得到顾客信息
    public static Customer getById(string id)
    {
        Customer customer = null;
        using (SqlHelper db = new SqlHelper())
        {
           //构建 SQL 语句
            string sql = "select CustomerName,LoginPassword,PayPassword from
            Customer where CustomerID=@id";
            using (DbCommand command = db.GetSqlStringCommond(sql))
            {
                db.AddInParameter(command, "@id", DbType.String, id);
                using (DbDataReader reader = db.ExecuteReader(command))
                {
                    //将从数据库读取的数据转换成 Customer 类的实例
                    if (!reader.Read())
                        return null;
                    customer = new Customer();
                    customer.id = id;
                    customer.name = reader["CustomerName"].ToString();
                    customer.loginPassword = reader["LoginPassword"].
                    ToString();
                    customer.payPassword = reader["PayPassword"].ToString();
                }
            }
        }
        return customer;
    }
}
```

（8）在数据访问层 BankDAL 项目中添加一个 BankAccountDal 类，完成银行账户相关的数据访问。本例中用到两个功能，一个功能是根据银行账号得到账号详情，另外一个功能是修改账户余额。

```
public static class BankAccountDal
{
    //根据账号得到账号详细信息
    public static BankAccount getById(string no)
    {
        BankAccount account = null;
        using (SqlHelper db = new SqlHelper())
        {
            //构建查询用的 select 语句
            string sql = "select CustomerID,Balance,IsLocked from BankAccount
            where AccountNo=@no";
            using (DbCommand command = db.GetSqlStringCommond(sql))
            {
                db.AddInParameter(command, "@no", DbType.String, no);
                using (DbDataReader reader = db.ExecuteReader(command))
                {
                    //将从数据库读取的数据转换成 BankAccount 类的实例
                    if (!reader.Read())
                        return null;
                    account = new BankAccount();
                    account.no = no;
                    account.customerID = reader["CustomerID"].ToString();
                    account.balance = Convert.ToDouble( reader["Balance"]);
                    account.isLocked = Convert.ToBoolean(reader["IsLocked"]);
                }
            }
        }
        return account;
    }
    /// <summary>
    /// 向账户中增加金额
    /// </summary>
    /// <param name="account">要增加金额的账户</param>
    /// <param name="money">增加的数额（负数则减少）</param>
    public static void addMoney(string account, double money)
    {
        using (SqlHelper db = new SqlHelper())
        {
            string sql = "update BankAccount set Balance=Balance+@money where
            AccountNo=@no";
            using (DbCommand command = db.GetSqlStringCommond(sql))
            {
                db.AddInParameter(command, "@no", DbType.String, account);
                db.AddInParameter(command, "@money", DbType.Double, money);
                db.ExecuteNonQuery(command);
            }
        }
    }
}
```

（9）在数据访问层 BankDAL 项目中添加一个 PolicyDal 类，完成对银行政策的数据访问功能。

```
public static class PolicyDal
{
    #region 常用政策 ID
    private const string ConstSingleTransferLimitID = "policy001";
    //单笔转账上限 ID
    private const string ConstTranserTimesLimitID = "policy002";
    //单日交易次数上限 ID
    private const string ConstTranserMoneyLimitID = "policy003";
    //单日转账总额上限 ID
    #endregion
    /// <summary>
    /// 根据政策 ID 得到政策内容
    /// </summary>
    /// <param name="id">政策 ID</param>
    /// <returns>政策内容</returns>
    public static BankPolicy getById(string id)
    {
        BankPolicy policy = null;
        using (SqlHelper db = new SqlHelper())
        {
            string sql = "select Descroption,PolicyContent from BankPolicy
            where PolicyID=@id";
            using (DbCommand command = db.GetSqlStringCommond(sql))
            {
                db.AddInParameter(command, "@id", DbType.String, id);
                using (DbDataReader reader = db.ExecuteReader(command))
                {
                    if (!reader.Read())
                        return null;
                    //将数据库中读取的数据转换成实体类 BankPolicy
                    policy = new BankPolicy();
                    policy.id = id;
                    policy.description = reader["Description"].ToString();
                    policy.content = reader["PolicyContent"].ToString();
                }
            }
        }
        return policy;
    }
    //得到单笔转账金额上限
    public static double getSingleTransferLimit()
    {
        BankPolicy policy = getById(ConstSingleTransferLimitID);
        return double.Parse(policy.content);
    }
    //得到单日转账交易次数上限
    public static int getDailyTransferTimesLimit()
    {
        BankPolicy policy = getById(ConstTranserTimesLimitID);
        return int.Parse(policy.content);
    }
    //得到单日累计转账金额上限
    public static double getDailyTransferMoneyLimit()
    {
        BankPolicy policy = getById(ConstTranserMoneyLimitID);
        return double.Parse(policy.content);
    }
}
```

（10）在数据访问层 BankDAL 项目中添加一个 TransactionDal 类，完成对转账交易明细的数据访问功能。

```
public static class TransactionDal
{
    //根据交易 ID 得到交易详情
    public static AccountTransaction getByID(int id)
    {
        AccountTransaction tran = null;
        using (SqlHelper db = new SqlHelper())
        {
            //构建查询用的 select 语句
            string sql = "select SourceAccount, DestinationAccount,
            TransactionDate, TransferMoney  from AccountTransaction where
            TransactionID=@id";
            using (DbCommand command=db.GetSqlStringCommond(sql))
            {
                db.AddInParameter(command, "@id", DbType.Int32, id);
                using (DbDataReader reader=db.ExecuteReader(command))
                {
                    if (!reader.Read())                //未找到此 ID
                        return null;
                   //根据读取的数据构建一个实体类
                    tran = new AccountTransaction();
                    tran.transactionID = id;
                    tran.sourceAccount = Convert.ToString(reader
                    ["SourceAccount"]);
                    tran.destinationAccount = Convert.ToString(reader
                    ["DestinationAccount"]);
                    tran.transactionDate = Convert.ToDateTime(reader
                    ["TransactionDate"]);
                    tran.transerMoney = Convert.ToDouble(reader
                    ["TransferMoney"]);
                }
            }
        }
        return tran;
    }
    /// <summary>
    /// 添加新的交易信息
    /// </summary>
    /// <param name="tran">交易信息</param>
    /// <returns>添加的行数</returns>
    public static int addTransaction(AccountTransaction tran)
    {
        using (SqlHelper db=new SqlHelper())
        {
            //构建插入 SQL 语句
            string sql = "insert into AccountTransaction "
                +"(SourceAccount,DestinationAccount,TransactionDate,
                TransferMoney) "
                +" values(@source,@dest,@date,@money)";
            //创建一个命令，并添加各个参数
            using (DbCommand command = db.GetSqlStringCommond(sql))
            {
                db.AddInParameter(command, "@source", DbType.String, tran.
                sourceAccount);
                db.AddInParameter(command, "@dest", DbType.String, tran.
```

```
                destinationAccount);
                db.AddInParameter(command, "@date", DbType.DateTime, tran.
                transactionDate);
                db.AddInParameter(command, "@money", DbType.Double, tran.
                transerMoney);
                return db.ExecuteNonQuery(command);      //返回受影响的行数
            }
        }
    }
    /// <summary>
    /// 根据账号和交易时间获得交易详情列表
    /// </summary>
    /// <param name="account">银行账号</param>
    /// <param name="from">查询的起始日期时间</param>
    /// <param name="to">查询的终止日期时间</param>
    /// <returns>符合条件的交易列表</returns>
    public static List<AccountTransaction> getByAccountAndDate(string
    account, DateTime from, DateTime to)
    {
        List<AccountTransaction> list = new List<AccountTransaction>();
        using (SqlHelper db = new SqlHelper())
        {
            //构建查询数据的 select 语句
            string sql = "select TransactionID, DestinationAccount,
            TransactionDate, TransferMoney "
                + " from AccountTransaction where (SourceAccount=@account) "
                +" and (TransactionDate between @date1 and @date2) ";
            using (DbCommand command = db.GetSqlStringCommond(sql))
            {
                //以下三条语句向查询语句添加参数
                db.AddInParameter(command, "@account", DbType.String,
                account);
                db.AddInParameter(command, "@date1", DbType.DateTime, from);
                db.AddInParameter(command, "@date2", DbType.DateTime, to);
                using (DbDataReader reader = db.ExecuteReader(command))
                {
                    //针对查询到的数据循环
                    //每条数据生成一个 AccountTransaction 类，添加到列表中
                    while (reader.Read())
                    {
                        var tran = new AccountTransaction();
                        tran.transactionID = Convert.ToInt32(reader
                        ["TransactionID"]);
                        tran.sourceAccount = account;
                        tran.destinationAccount = Convert.ToString( reader
                        ["DestinationAccount"]);
                        tran.transactionDate = Convert.ToDateTime(reader
                        ["TransactionDate"]);
                        tran.transerMoney = Convert.ToDouble(reader
                        ["TransferMoney"]);
                        list.Add(tran);
                    }
                }
            }
        }
        return list;
    }
}
```

（11）在业务逻辑层 BankBLL 项目中添加一个 CustomerBll 类，实现银行顾客相关的业务逻辑。由于本例中没有与顾客相关的业务逻辑，所以 CustomerBll 类仅是调用数据访问层 CustomerDal 的相关功能。

```
public static class CustomerBll
{
    public static Customer getById(string id)
    {
        return CustomerDal.getById(id);
    }
}
```

（12）在业务逻辑层 BankBLL 项目中添加一个 BankAccountBll 类，实现银行账户相关的业务逻辑。本例中与银行账户相关的一条业务规则有一条，当银行账户被冻结后，则不能修改其账户余额。

```
public static class BankAccountBll
{
    public static BankAccount getById(string no)
    {
        return BankAccountDal.getById(no);
    }
    public static void addMoney(string account, double money)
    {
        BankAccount theAccount = BankAccountBll.getById(account);
        if (theAccount.isLocked)
            throw new BankException("账户["+account+"]被冻结，不能执行此操作。");
        BankAccountDal.addMoney(account, money);
    }
}
```

以上语句中用到的一个异常类 BankException 为自定义异常类，代码如下：

```
//银行业务异常类
[Serializable]
public class BankException : ApplicationException
{
  public BankException() { }
  public BankException( string message ) : base( message ) { }
  public BankException( string message, Exception inner ) : base( message, inner )
  { }
  protected BankException(
     System.Runtime.Serialization.SerializationInfo info,
     System.Runtime.Serialization.StreamingContext context ) : base( info,
     context ) { }
}
```

（13）在业务逻辑层 BankBLL 项目中添加一个 PolicyBll 类，实现银行政策相关的业务逻辑。由于本例没有具体与银行政策相关的业务规则，PolicyBll 类的作用仅是调用数据访问层相应的方法。

```
public static class PolicyBll
{
    public static double getSingleTransferLimit()
    {
        return PolicyDal.getSingleTransferLimit();
    }
    public static int getDailyTransferTimesLimit()
```

```
    {
        return PolicyDal.getDailyTransferTimesLimit();
    }
    public static double getDailyTransferMoneyLimit()
    {
        return PolicyDal.getDailyTransferMoneyLimit();
    }
}
```

（14）在业务逻辑层 BankBLL 项目中添加一个 TransactionBll 类，实现银行转账相关的业务逻辑。本例中，银行转账有复杂的业务逻辑，如单笔转账金额封顶、当日累计转账金额封顶、当日累计转账次数最大值等。在 TransactionBll 类中要实现所有这些业务规则。

```
public static class TransactionBll
{

    public static AccountTransaction getByID(int id)
    {
        return TransactionDal.getByID(id);
    }
    public static List<AccountTransaction> getByAccountAndDate(string
    account, DateTime from, DateTime to)
    {
        return TransactionDal.getByAccountAndDate(account, from, to);
    }
    /// <summary>
    /// 执行银行转账业务
    /// </summary>
    /// <param name="customerId">顾客 ID</param>
    /// <param name="from">源账户</param>
    /// <param name="to">目的账户</param>
    /// <param name="toName">目的账户的顾客名称</param>
    /// <param name="money">转账金额</param>
    public static void transferMoney(string customerId,string from, string
    to, string toName,double money)
    {
        BankAccount source = BankAccountBll.getById(from);
        //得到源账户信息
        if (source.customerID != customerId)
            throw new BankException("顾客不拥有所请求账户。");
        if (source.balance < money)
            throw new BankException("账户余额不足");
        var list = TransactionBll.getByAccountAndDate(from, DateTime.Today,
        DateTime.Today.AddDays(1).AddSeconds(-1));
        int n = PolicyBll.getDailyTransferTimesLimit();
        if(list.Count>=n)
            throw new BankException("账户今天交易次数已经达到上限");
        if(money<=0)
            throw new BankException("转账金额必须大于零");
        double d = PolicyBll.getSingleTransferLimit();
        if(money>=d)
            throw new BankException("转账金额超出单笔转账上限");
        d = list.Sum(tr => tr.transerMoney);
        //计算今天转账总额
        if(d+money>PolicyBll.getDailyTransferMoneyLimit())
            throw new BankException("此次转账将使账户今天转账总金额将超过上限，转
            账失败");
        BankAccount dest = BankAccountBll.getById(to);
```

```
                                        //得到目的账户信息
        string name = CustomerBll.getById(dest.customerID).name;
        if(name!=toName)
            throw new BankException("目的账户名称不对。");
        //将所有数据修改操作（增加余额、减少余额、记录交易明细）封装在一个事务中
        using (System.Transactions.TransactionScope scope = new System.
        Transactions.TransactionScope())
        {
            BankAccountBll.addMoney(from, -money);   //减少源账户余额
            BankAccountBll.addMoney(to, money);      //增加目的账户余额
            AccountTransaction tran = new AccountTransaction();
            tran.sourceAccount = from;
            tran.destinationAccount = to;
            tran.transerMoney = money;
            tran.transactionDate = DateTime.Now;
            TransactionDal.addTransaction(tran);     //记录交易信息
            scope.Complete();                        //成功结束事务
        }
    }
}
```

（15）在表现层 ASP.NET Web 应用程序 BankApp 中，添加一个页面 TierBankPage.aspx，在页面上放置相应的控件供用户输入数据。

```
<h3>银行转账程序（三层结构版）</h3>
<table><tr>
<td>客户编号：<asp:TextBox ID="customer" runat="server"></asp:TextBox></td>
<td>登录密码：<asp:TextBox ID="loginPass" runat="server" TextMode="Password">
</asp:TextBox></td>
<td>支付密码：<asp:TextBox ID="payPass" runat="server" TextMode="Password">
</asp:TextBox></td>
</tr><tr>
<td>转出账号：<asp:TextBox ID="source" runat="server"></asp:TextBox></td>
 <td>转账金额：<asp:TextBox ID="money" runat="server"></asp:TextBox></td>
<td></td></tr>
<tr>
<td> 接收账号：<asp:TextBox ID="destination" runat="server"></asp:TextBox>
</td>
<td> 账号名称：<asp:TextBox ID="destinationName" runat="server"></asp:TextBox>
</td>
<td><asp:Button ID="ok" runat="server" Text="转账" Width="100" onclick=
"ok Click" />
</td></tr>
</table>
<asp:Label runat="server" ID="error" ForeColor="Red" Visible="false">
</asp:Label>
<asp:Label runat="server" ID="message" Visible="false" ></asp:Label>
```

（16）在 TierBankPage.aspx 页面中”转账”按钮的 Click 事件中，调用业务逻辑层相应功能实现银行转账，如果发生错误，则显示错误提示。

```
protected void ok_Click(object sender, EventArgs e)
{
    var c = CustomerBll.getById(customer.Text);            //得到顾客信息
```

```
    if (c == null)
    {
        showError("用户不存在。");
        return;
    }
    if (c.loginPassword != loginPass.Text || c.payPassword != payPass.Text)
        showError("密码不正确。");
    try
    {
        //调用业务逻辑层类实现转账功能
        BankBLL.TransactionBll.transferMoney(customer.Text,source.Text,
            destination.Text, destinationName.Text, double.Parse(money.
             Text));
        showMessage("转账操作成功。");
    }
    catch(BankEntity.BankException ex)                     //发生自定义异常
    {
            showError(ex.Message);
    }
    catch (Exception ex)                                   //发生其他异常
    {
        showError("发生未知错误。"+ex.Message);
    }
}
private void showError(string errorMessage)                //显示错误信息
{
    error.Text = errorMessage;
    error.Visible = true;
    message.Visible = false;
    message.Text = "";
}
private void showMessage(string content)
                                                          //显示交易成功提示
{
    message.Visible = true;
    error.Visible = false;
    error.Text = "";
    message.Text = content;
}
```

5.3.6　三层结构程序的优势

对比未分层的银行转账程序和三层结构的银行转账程序，三层结构有如下优势。

（1）功能内聚。在三层结构的银行转账与数据库相关的代码集中在数据访问层，而且涉及具体每个表的操作都集中在一个类中，其他各层也具有相同特点。这种设计使得类和项目的功能具有很高的内聚性，符合页面对象设计原则。

（2）代码复用。由于数据访问和业务逻辑都封装特定的类和方法中，存储于类库中，无论在任何地方（如不同的页面、甚至不同的项目中）需要这些功能，都可以添加对类库的引用，调用相应方法即可，遵守了一个功能只实现一次的原则。

（3）不易修改。由于相似功能在一个类中集中存放，而不是分散于整个项目众多的页面后台代码中，使得代码修改更加简单。例如，假设数据库中银行顾客表的某个字段名称发生了改变，那么只需要修改数据访问层的 CustomerDal 类中的相关代码即可，数据访

问层其他类、业务逻辑层、表现层代码都不用作任何改动。

（4）文件规模适中。由于将整个程序功能分成三层，使得每一部分的功能相对简单，限制了文件规模，不会出现不易阅读和维护的庞大源码文件。

（5）强类型。由于业务实体类封装数据库中数据，用类的属性表示数据表中的字段，从而实现了强类型，保证了类型安全，还可以充分利用 Visual Studio 的智能提示功能，提高编程效率。例如，当使用 DataTable 封装 BankAccount 表中的数据时，如果要访问账户余额，则需要使用以下语句：

```
double balance=Convert.ToDouble(talbe.Rows[0]["Balance"]);
```

上述语句中，需要记住数据表中的字段名称和类型，还需要进行类型转换。如果字段名称记错或者打字时出现拼写错误，由于字段名在字符串中，Visual Studio 不会进行语法检测，因此识别不出错误的字段名。

在三层结构中，用实体类 BankAccount 封装了 BankAccount 表中数据，同样如果要访问账户余额，则需要以下代码。

```
BankAccount account=BankAccountDal.getByID("账号");
double balance=account.balance;
```

上述代码 BankAccount 类及 balance 属性都是强类型，在 Visual Studio 开发环境中，能够自动提示 balance 属性并进行类型检测，提高了编程效率，避免了类型错误。

5.4 单 元 测 试

单元测试又称为模块测试，是针对程序模块（如方法）来进行正确性检验的测试工作。单元测试主要是用来检验程序的内部逻辑，通常由撰写代码的编写者负责进行。当开发人员编写完成程序中的一个功能（在.NET 中此功能通常对应于一个方法）后，应该写一段测试代码以验证其正确性，这个工作就是单元测试。

5.4.1 单元测试概述

单元测试是一项很重要的工作，有助于尽早发现程序中的 bug。根据软件工程理论，软件中 bug 发现的时间越晚，修改这个 bug 所花费的代价就越高，所以应该尽早发现程序中的错误并及时修改。用于单元测试的代码也是整个软件的一个重要组成部分，需要妥善保存并在后续开发过程中使用。实践证明，对于程序某个地方的修改可能会在无意中影响到其他地方从而导致程序出错。因此，在对软件进行了较大的改动以后，不但应该测试被修改的代码，还要对与之相关的所有代码都重新执行一遍测试，以确保程序的修改不会引入新的 bug。

单元测试是软件开发过程中一个必不可少的环节，对于保证软件质量有着重要作用，这一点在测试驱动开发（Test-driven development）中体现较为明显。测试驱动开发是一种现代的软件开发方法，利用测试来驱动软件程序的设计和实现。测试驱动开发是极限编程中倡导的程序开发方法，其指导思想简单来说就是在编写实现一个功能以前先写测试程序，

然后再编码使其通过测试。

软件开发有两个基本的衡量指标：实现的功能和质量。测试驱动开发就像两顶帽子思考法的开发方式，先戴上实现功能的帽子，在测试的辅助下，快速实现正确的功能；再戴上重构的帽子，在测试的保护下，通过去除冗余和重复的代码，提高代码重用性，实现对质量的改进。可见测试在测试驱动开发中确实属于核心地位，贯穿了开发的始终。

5.4.2　创建和运行单元测试

在 Visual Studio 中可以完成包括单元测试在内的各种测试。测试也是一种代码，通过执行这些测试代码调用被测试的模块，并根据执行结果判断测试是否通过。进行测试时经常用到以下几个类。

- ❑ TestClassAttribute 类：这是一个属性（Attribute）类，所有的测试类都必须添加这个属性。
- ❑ TestMethodAttribute 类：这也是一个属性（Attribute）类，所有的需要独立执行的测试方法都必须添加这个属性。
- ❑ TestContext 类：表示测试上下文，用于存储提供给单元测试的信息。
- ❑ Assert 类：该类包含计算布尔值条件的一组静态方法，如 AreEqual、IsTrue、IsNull 等。如果此条件计算为 true，则断言通过。如果所验证的条件不为 true，则断言将失败，测试不通过。

下面以 5.3 节所创建的三层结构的银行转账程序为例，说明创建和运行单元测试的步骤。由于单元测试的方法基本相同，此处只选择数据访问层中几个比较典型的类和方法进行测试。

（1）在 Visual Studio 中打开银行转账程序解决方案，打开 BankDAL 项目中的 CustomerDal 类，在类名上右击，从弹出的快捷菜单中选择”创建单元测试”选项，如图 5.22 所示。

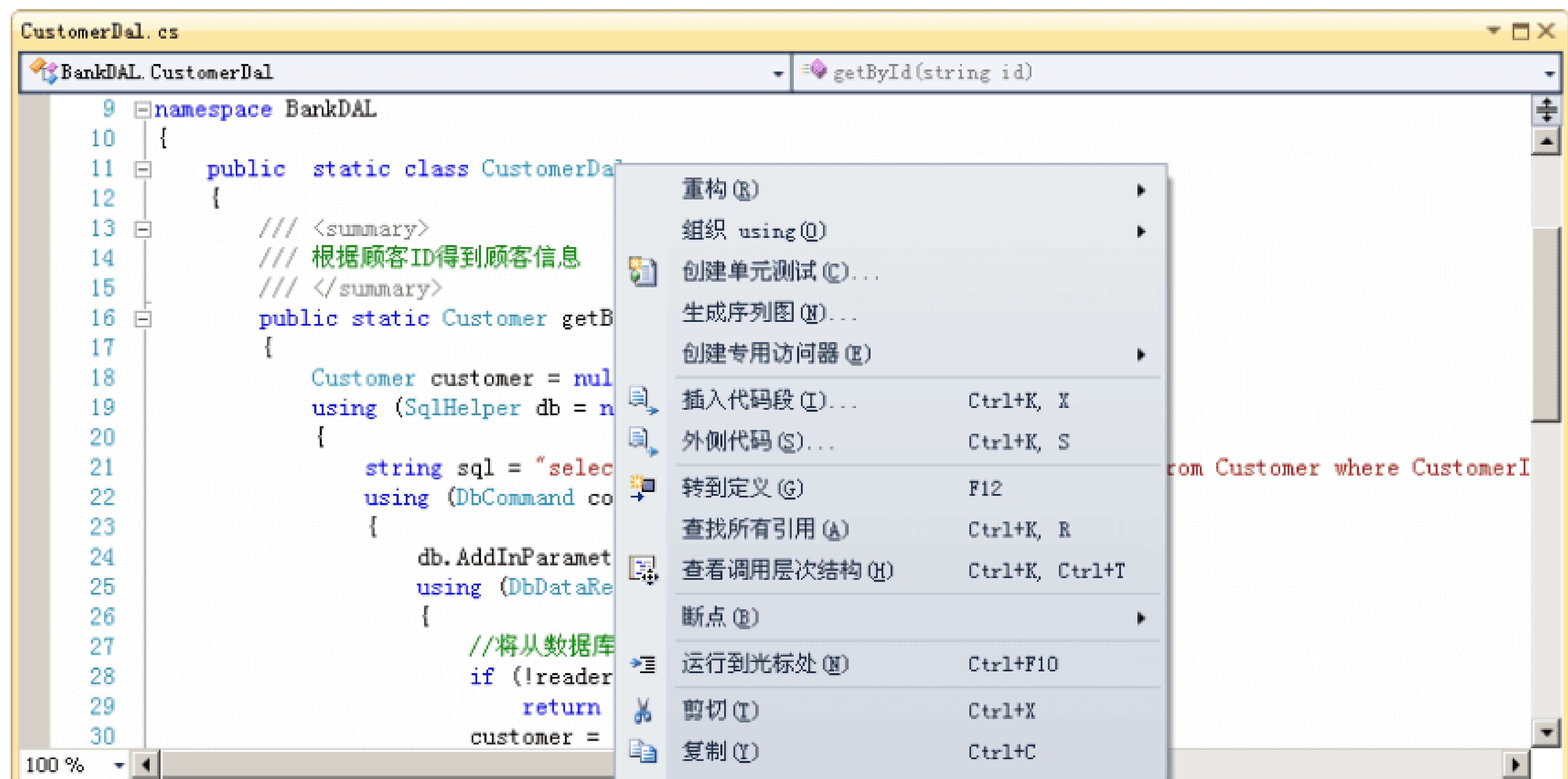


图 5.22　创建单元测试

（2）之后会弹出如图 5.23 所示的“创建单元测试”对话框，对话框中自动选中了 CustomerDal 类中所有方法，Visual Studio 将会为选中的方法自动生成单元测试代码框架。

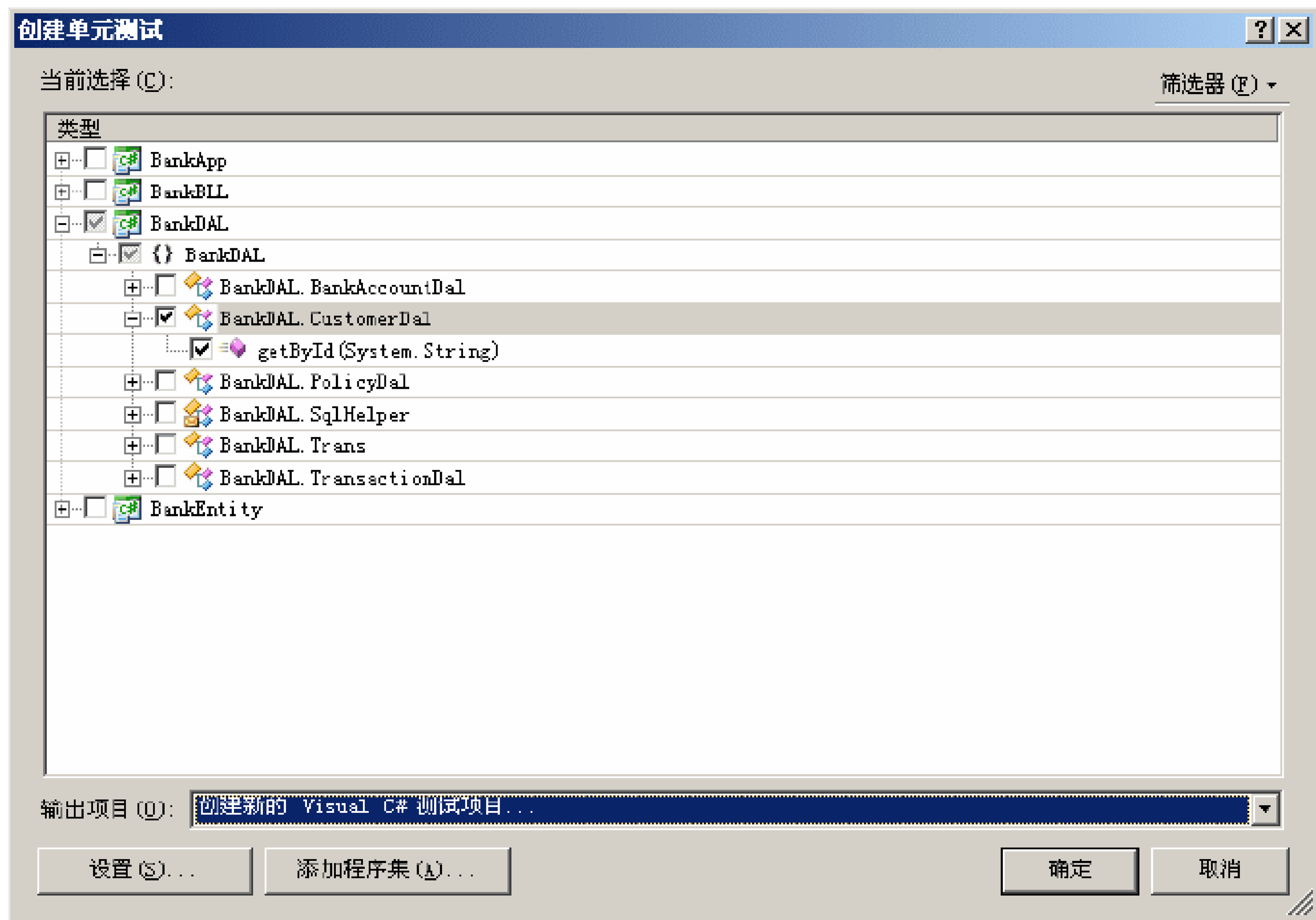


图 5.23 “创建单元测试”对话框

（3）单击“确定”按钮，则 Visual Studio 会自动生成一个 CustomerDalTest 类，类中包含主要代码如下。

提示： 如果是第一次在解决方案中添加测试，Visual Studio 会添加一个新的测试项目，并将新生成的测试类添加到测试项目中。如果不是第一次添加测试，则会把测试类添加到已有的测试项目中。

```
 [TestClass()]                                              //说明这是一个测试类
public class CustomerDalTest
{
   private TestContext testContextInstance;                 //测试上下文
   public TestContext TestContext
   {
      get
      {
         return testContextInstance;
      }
      set
      {
         testContextInstance = value;
      }
   }
   [TestMethod()]                                           //说明这是一个测试方法
   public void getByIdTest()                                //测试方法框架
   {
      string id = string.Empty; // TODO: 初始化为合适的值
      Customer expected = null; // TODO: 初始化为合适的值
      Customer actual;
      actual = CustomerDal.getById(id);
      Assert.AreEqual(expected, actual);
      Assert.Inconclusive("Verify the correctness of this test method.");
   }
}
```

（4）修改自动生成的测试方法 getByIdTest 为以下内容。

```
[TestMethod()]
public void getByIdTest()
{
    string id = "110101198001011234";
    Customer actual;
    actual = CustomerDal.getById(id);                  //调用被测试方法
    Assert.IsNotNull(actual);                          //断言返回结果非空
    //以下代码将 Customer 对象的各个字段与数据库中数据进行比较，以验证其正确性
    Assert.AreEqual(id, actual.id);
    Assert.AreEqual("路人甲", actual.name);
    Assert.AreEqual("123456", actual.loginPassword);
    Assert.AreEqual("12345600", actual.payPassword);
}
```

（5）将 BankApp 项目中的配置文件和 App_Data 目录下的数据库文件复制到测试项目 BankDalTest 中。

提示： 测试运行时，会从测试项目而非被测试项目中寻找配置文件。因此，如果项目中用到配置文件，一定要记住把同样的配置文件复制到测试项目中。
如果项目使用 Sql Server 2005 Express 版数据库，并且连接字符串使用了相对路径引用数据库（如 AttachDbFileName=[DataDirectory]|bank.mdf），则测试时需要将数据库文件复制到测试项目中。

（6）右击 getByIdTest 方法内部，从弹出的快捷菜单中选择“运行测试”选项，如图 5.24 所示。

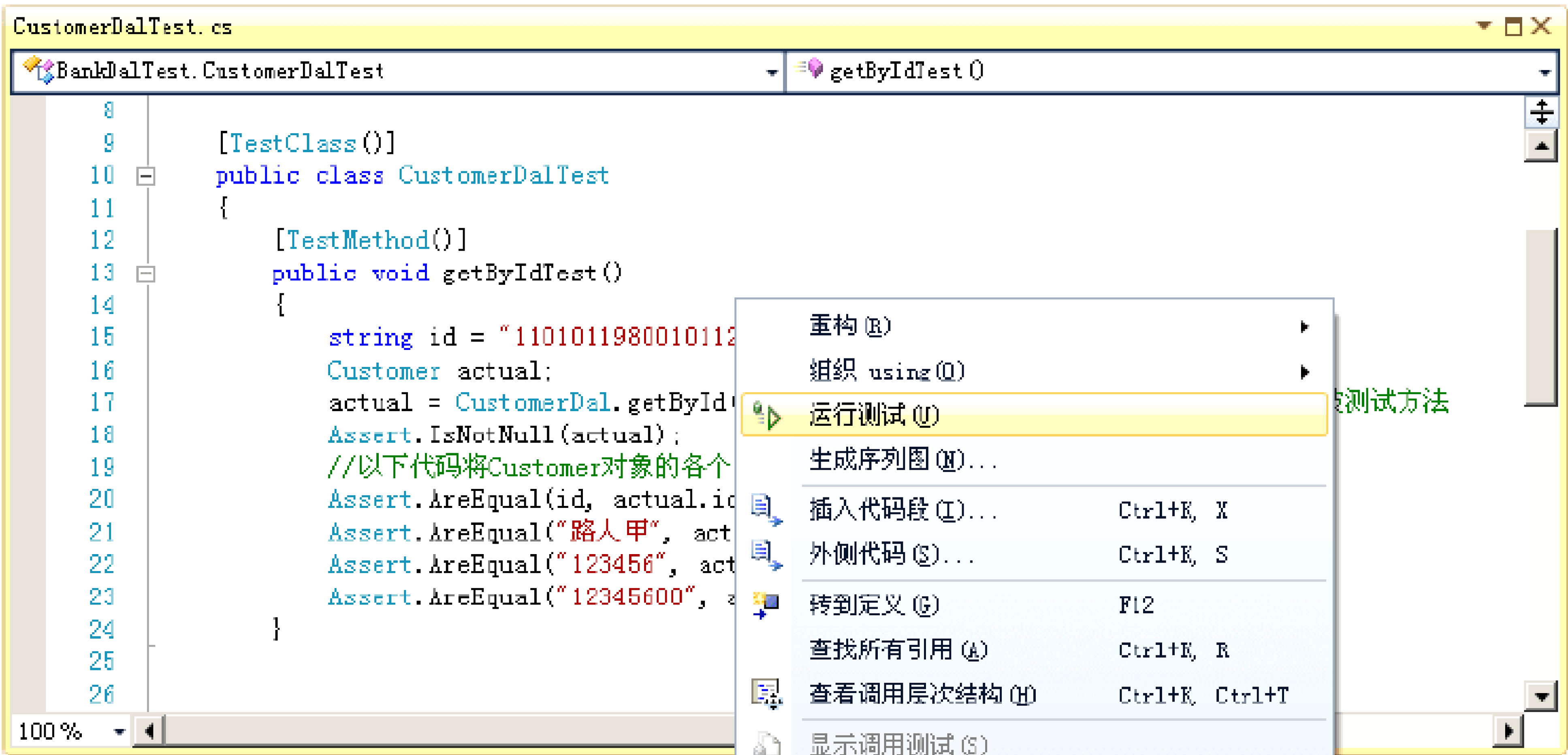


图 5.24　运行测试

（7）运行测试后，会在 Visual Studio 开发环境底部“测试结果”视图中显示测试运行结果，显示通过与否。如图 5.25 所示为 getByIdTest 测试通过时的情况。

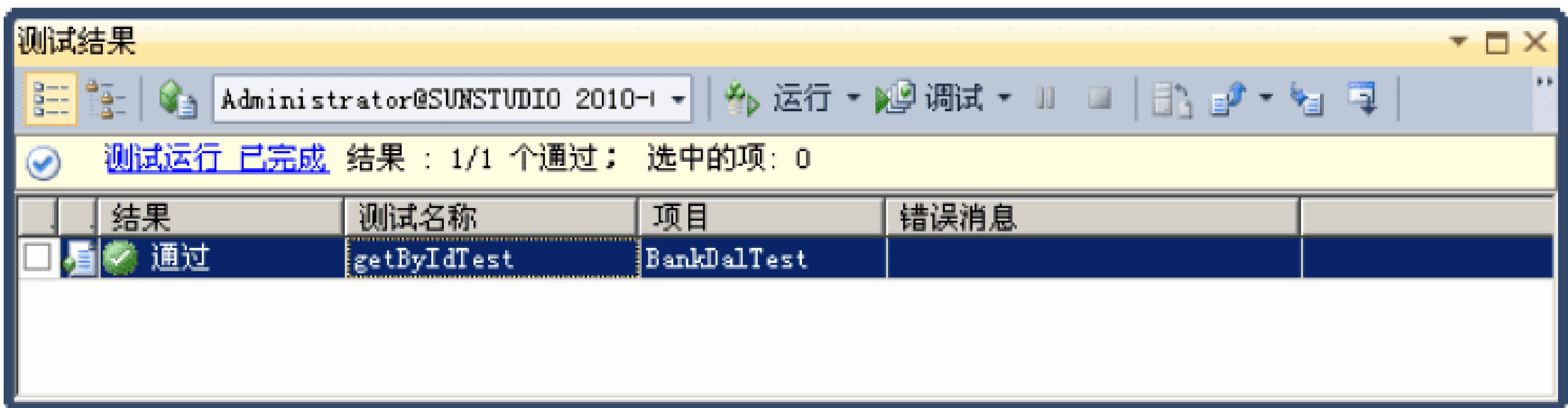


图 5.25　测试通过

（8）为了演示测试不通过的情况，修改 CustomerDal.getCustomerById 中的代码以产生个错误，例如将以下 SQL 语句中的 CustomerID 字段名故意写错为 ID。

```
select  CustomerName,LoginPassword,PayPassword  from  Customer  where
CustomerID=@id
```

再次运行测试，则测试结果如图 5.26 所示。

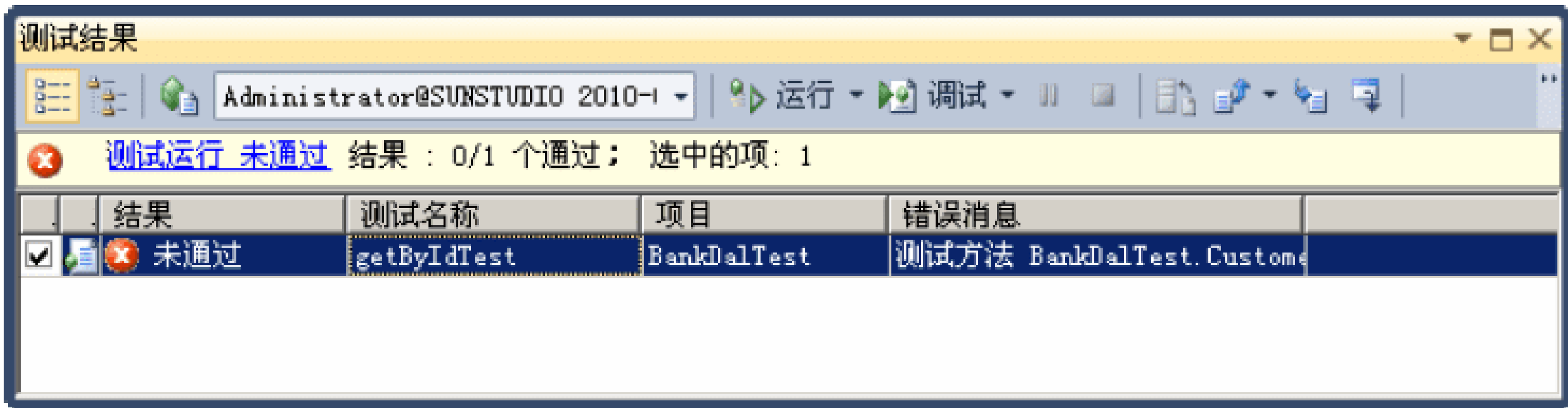


图 5.26　测试失败

在测试结果中双击未通过的测试，则会显示测试详情及未通过原因，如图 5.27 所示。

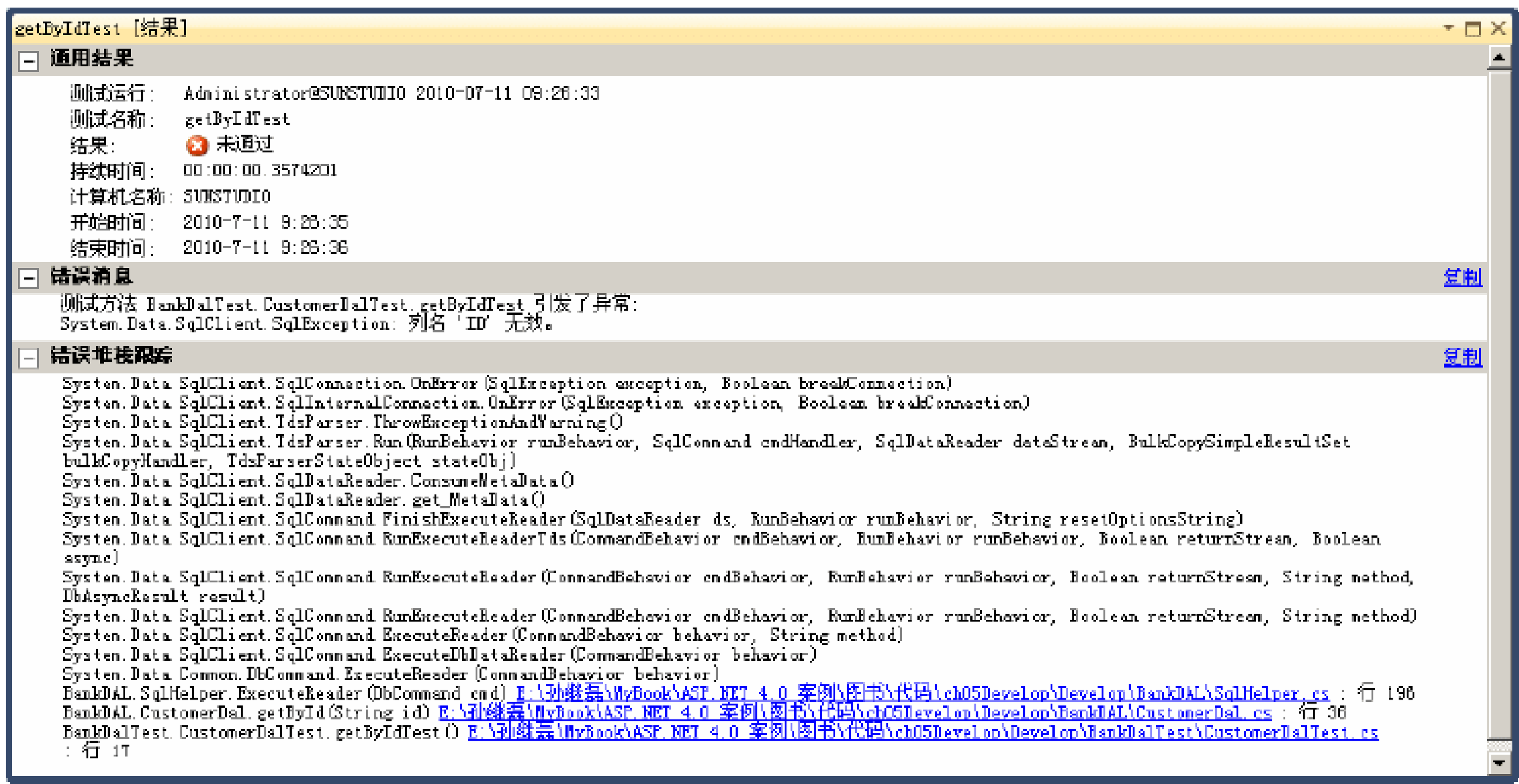


图 5.27　测试详情

从图 5.27 中可以看到测试未通过的原因是运行过程中产生了 SqlException 异常“列名 ID 无效”，并且能够看到异常发生的代码所在行，从而能够较容易找到并改正错误。

（9）要测试一个方法，应该测试其在各种情况下的执行情况，例如，当接受正常参数和非法参数时。对 getCustomerById() 方法也要用各种情况进行测试。在测试类 CustomerDalTest 中再添加一个方法，代码如下。

```
[TestMethod]
public void getByIdTest2()
{
    string id = "Not Exists";                    //数据库中不存在此 ID
    Customer actual;                             //断言返回结果为空
    actual = CustomerDal.getById(id);            //调用被测试方法
    Assert.IsNull(actual);
}
```

（10）运行新的测试并使其通过，如果有错误就修改代码。

5.4.3　管理单元测试

在真实的软件项目中，会产生大量测试类和测试方法，应该采用有效的组织方式管理这些测试。Visual Studio 中的测试列表是一种很方便的管理测试的方法。有些测试之间存在某种关系，例如同属于对同一个数据访问类的测试，或者都是与某一功能相关的测试，可以按照这些关系将一组测试组合在一起，形成测试列表。下面演示了创建和使用测试列表的具体步骤。

（1）从 Visual Studio 菜单中选择“测试”|“创建新的测试列表”命令，则弹出如图 5.28 所示的对话框，在其中输入测试列表名称，并选择测试列表位置。

（2）单击 OK 按钮后，会打开如图 5.29 所示的“测试列表编辑器”窗口。

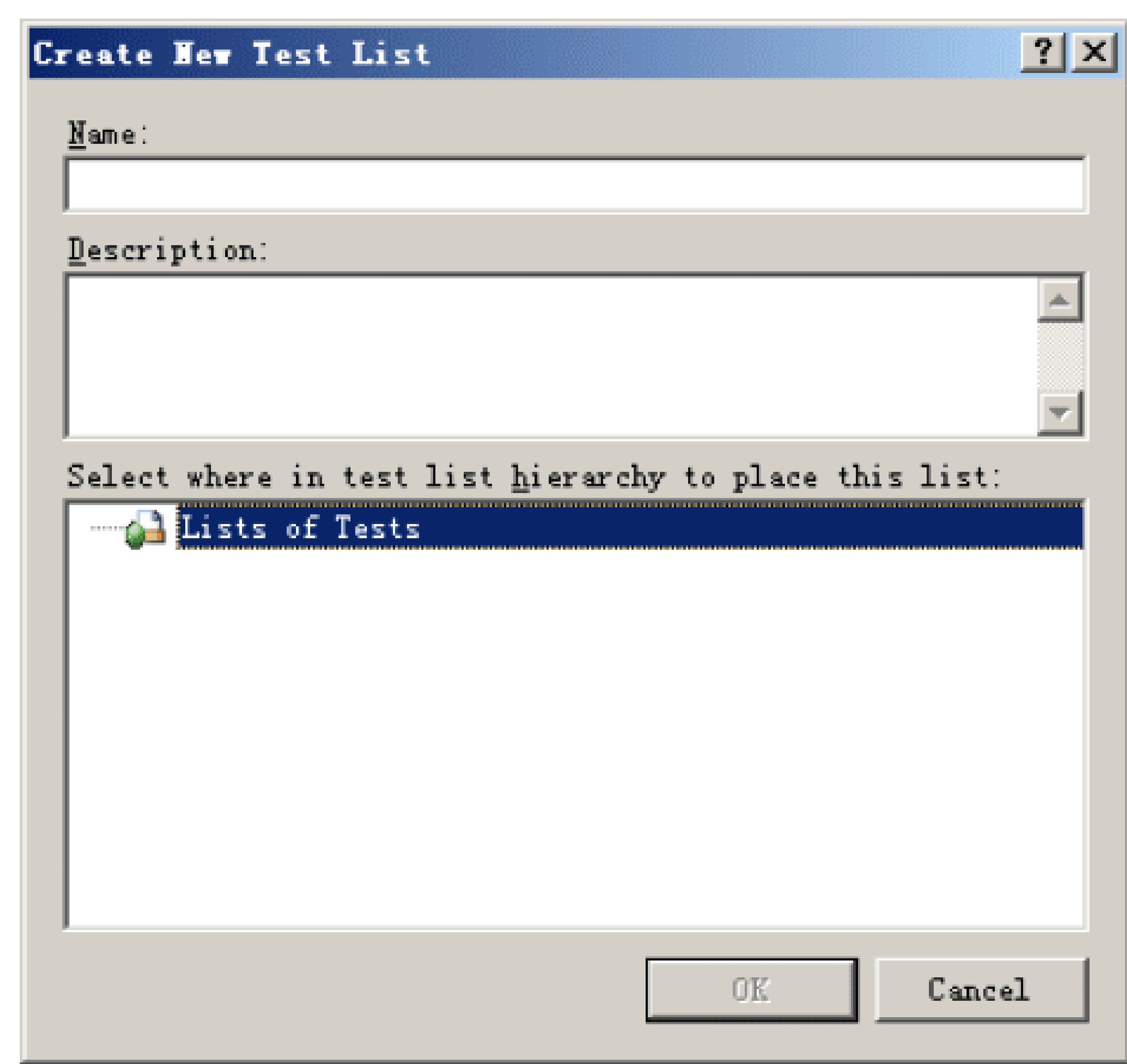


图 5.28　创建新的测试列表

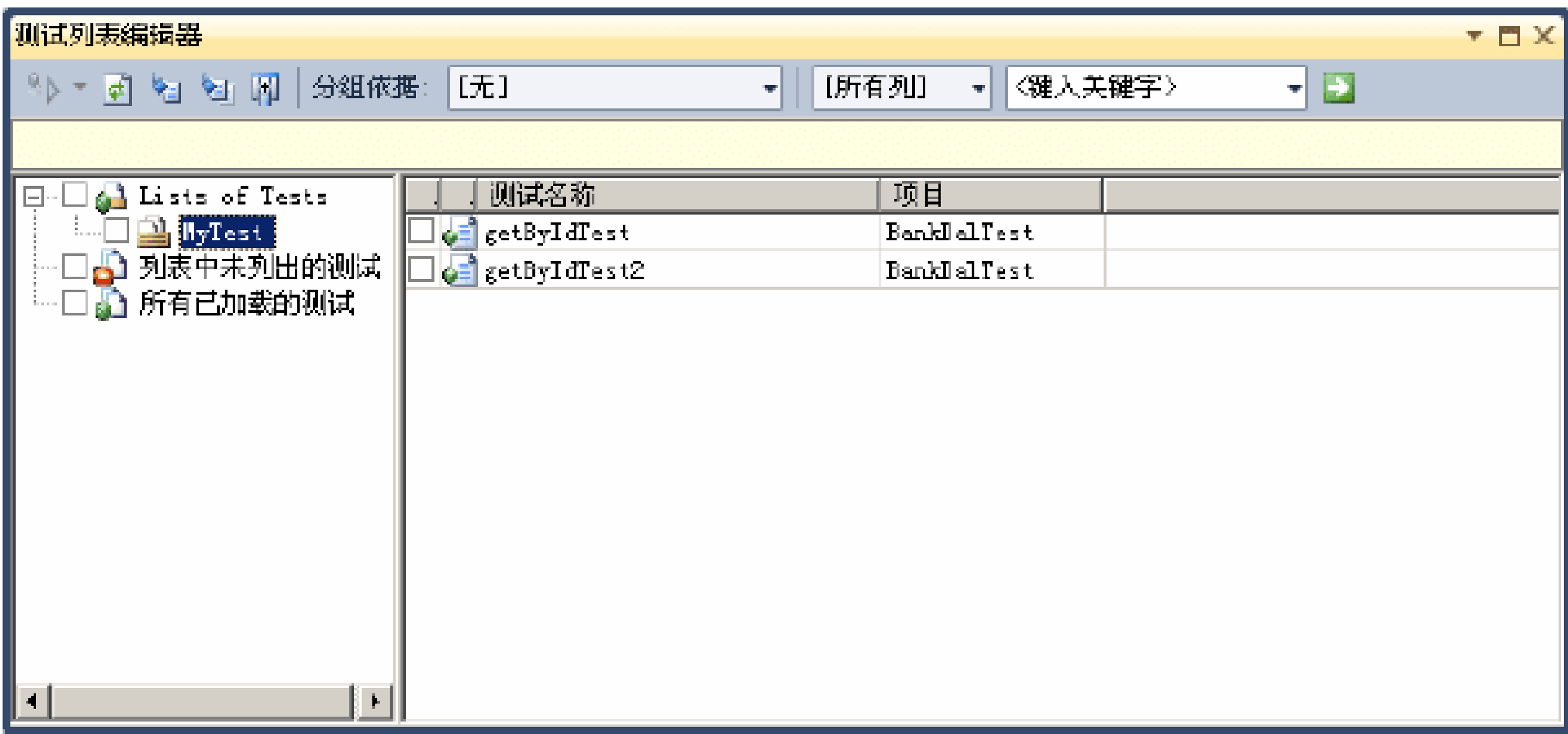


图 5.29　测试列表编辑器

（3）在“测试列表编辑器”窗口中，从左侧选择“列表中未列出的测试”，则右侧会列出不在当前测试列表中的测试，可以将测试拖动到左侧的一个测试列表中。本例将 CustomerDalTest 类中的两个测试 getByIdTest 和 getByIdTest2 拖动到 MyTest 列表中，如图 5.30 所示。

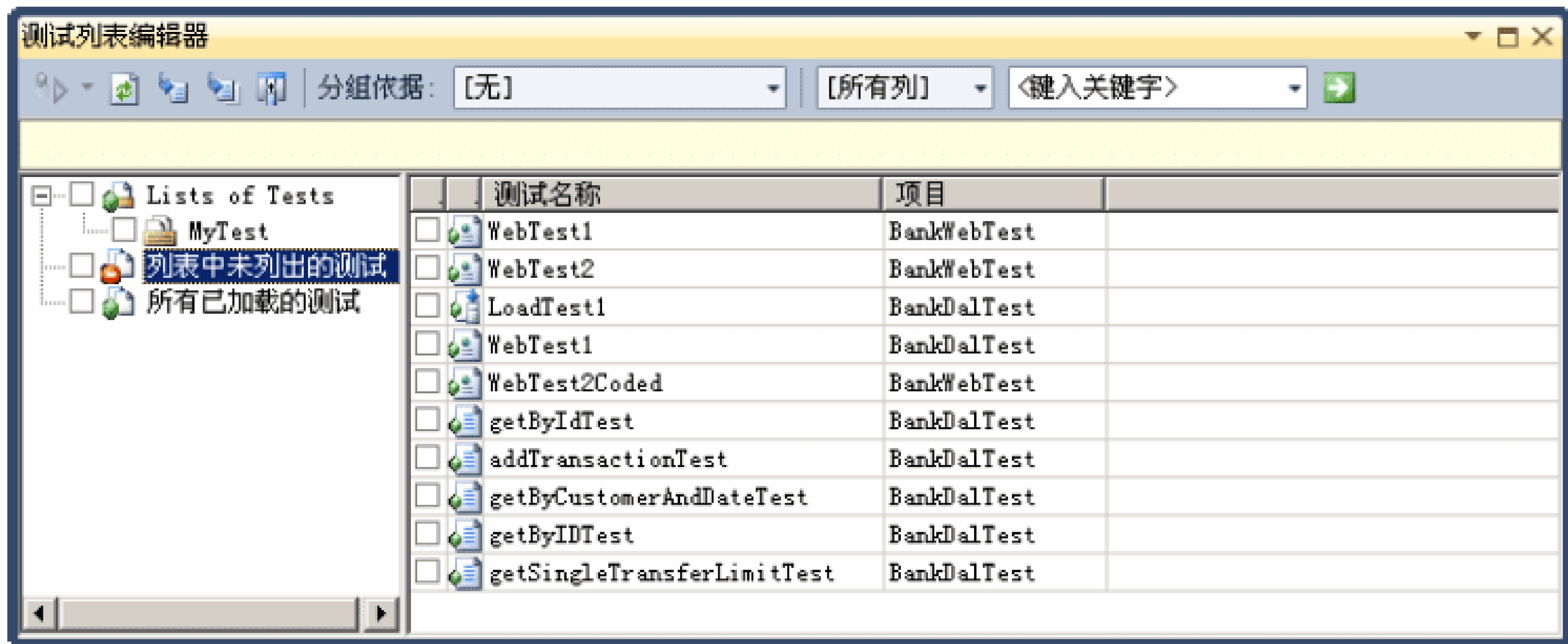


图 5.30　将测试加入测试列表

（4）从“测试列表编辑器”窗口右侧选中需要运行的测试，并单击工具栏上的运行测试按钮 即可运行所选中的测试。

5.4.4　代码覆盖率

为了提高测试可靠性，测试需要执行到目标方法中的所有代码。例如，如果目标方法中有 if…else…语句，则测试时 if 和 else 分支中的代码都应该被测试到。Visual Studio 中经过配置以后可以显示测试覆盖率，具体操作步骤如下。

（1）在解决方案资源管理器中，打开 Solution Items 文件夹，双击 Local.testsettings 项目，如图 5.31 所示。

（2）之后会打开如图 5.32 所示的“测试设置”对话框。在对话框的左侧列表中选择“数据和诊断”选项，从右侧选中“代码覆盖率”复选框，如图 5.32 所示，然后单击“应用”按钮。



图 5.31　local.testsettings 所在位置　　　　图 5.32　测试设置

（3）在图 5.32 所示的“测试设置”对话框的“数据和诊断”页面中，双击“代码覆盖”所在行，则弹出如图 5.33 所示的“代码覆盖明细”对话框，在其中可以选择要启用代码覆盖分析的项目，本例中选择 BankDAL 项目。

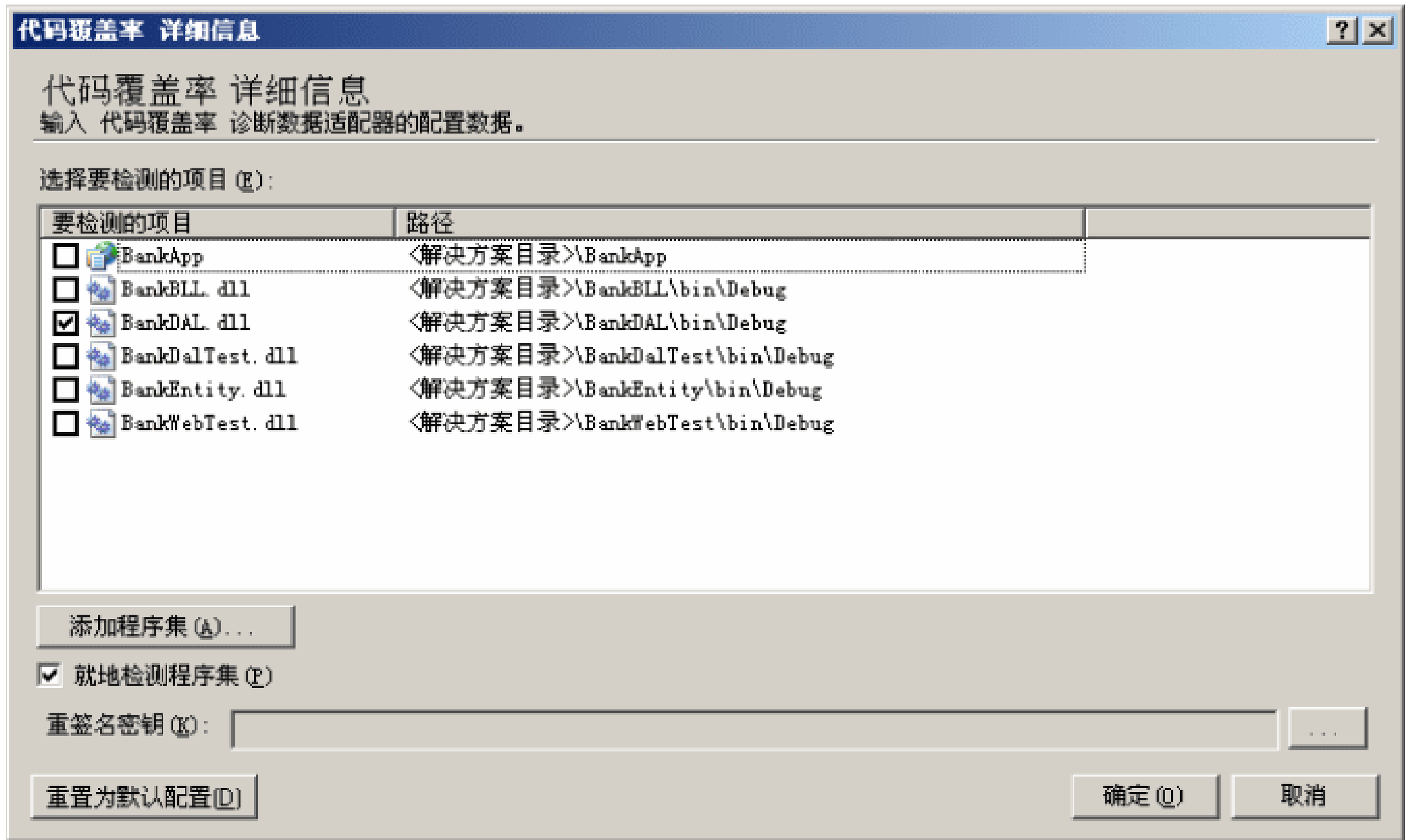


图 5.33　代码覆盖明细

（4）设置完成代码覆盖以后，选择几个测试并运行，测试运行完成后在测试结果试图中右击，从弹出菜单中选择“代码覆盖率结果”，如图 5.34 所示。

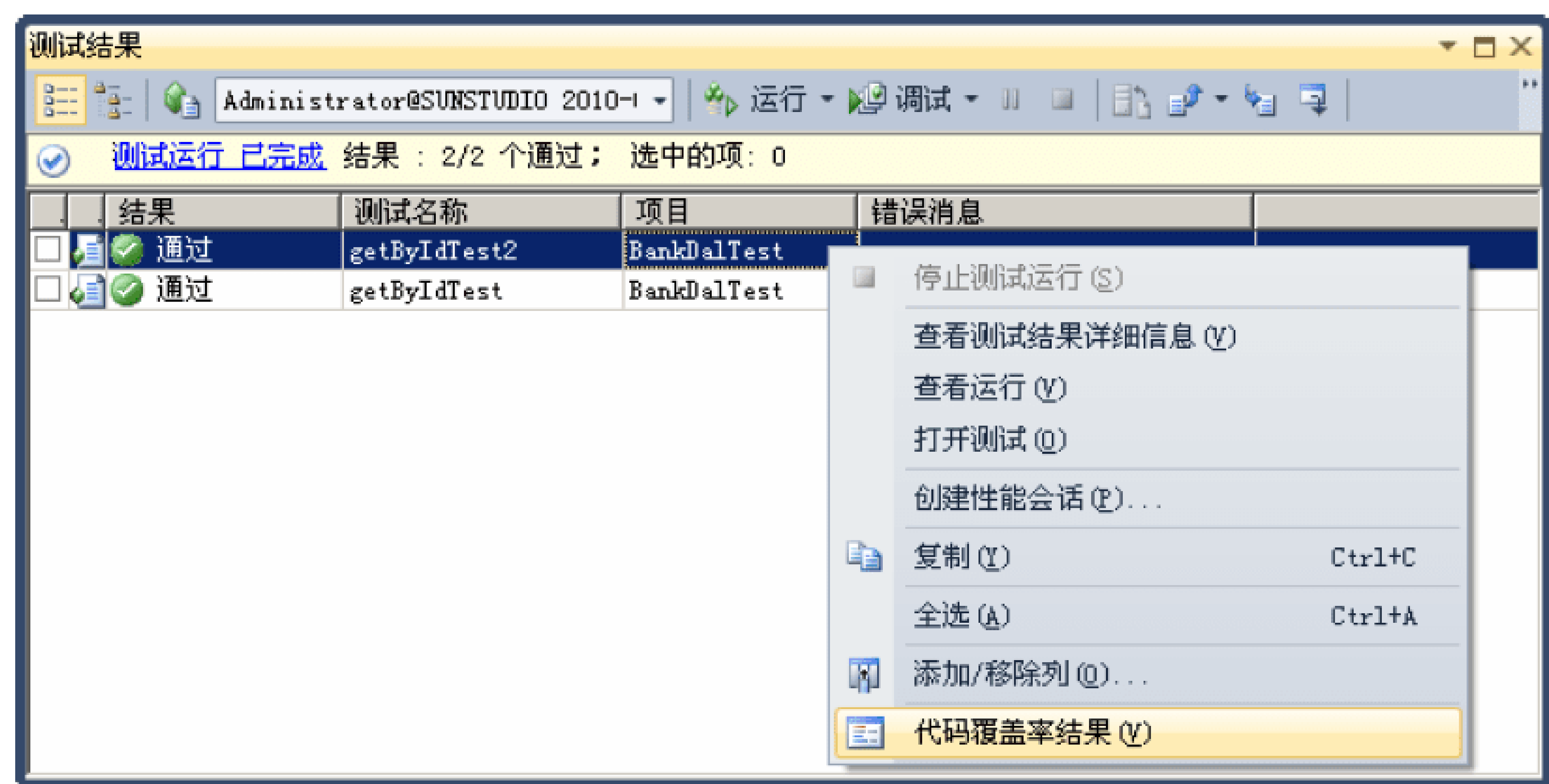


图 5.34　查看代码覆盖率结果

（5）代码覆盖率结果显示如图 5.35 所示，其中列出了被测试项目中所有类和方法，以及测试覆盖到和未覆盖到的代码数量和比例。根据这个结果，测试人员可以有针对性地调整自己的测试代码，提高代码覆盖率。

代码覆盖率结果

Administrator@SUNSTUDIO 2010-07-11 09:

层次结构	未覆盖(块)	未覆盖(% 块)	已覆盖(块)	已覆盖(% 块)
Administrator@SUNSTUDIO 2010-07-...	276	80.70%	66	19.30%
BankDAL.dll	276	80.70%	66	19.30%
{} BankDAL	276	80.70%	66	19.30%
BankAccountDal	36	100.00%	0	0.00%
CustomerDal	0	0.00%	26	100.00%
getById(string)	0	0.00%	26	100.00%
PolicyDal	38	100.00%	0	0.00%
SqlHelper	97	70.80%	40	29.20%
.cctor()	0	0.00%	7	100.00%
.ctor()	0	0.00%	3	100.00%
.ctor(string)	3	100.00%	0	0.00%
AddInParameter(class...	0	0.00%	8	100.00%
AddOutParameter(clas...	8	100.00%	0	0.00%
CreateConnection()	5	100.00%	0	0.00%
CreateConnection(str...	0	0.00%	5	100.00%
Dispose()	0	0.00%	2	100.00%

图 5.35　代码覆盖率结果

（6）Visual Studio 还支持以不同颜色显示测试覆盖到和未覆盖到的代码，让开发人员对代码覆盖结果一目了然。例如，在图 5.35 中，看到 CustomerDal 类的代码覆盖率为 42.31%，如果想查看具体哪些代码没有覆盖，则在图 5.35 所示的代码覆盖视图中双击 CustomerDal 类，会打开 CustomerDal.cs 文件，并分别用浅蓝色和浅棕色背景显示覆盖到和未覆盖到的代码，如图 5.36 所示。

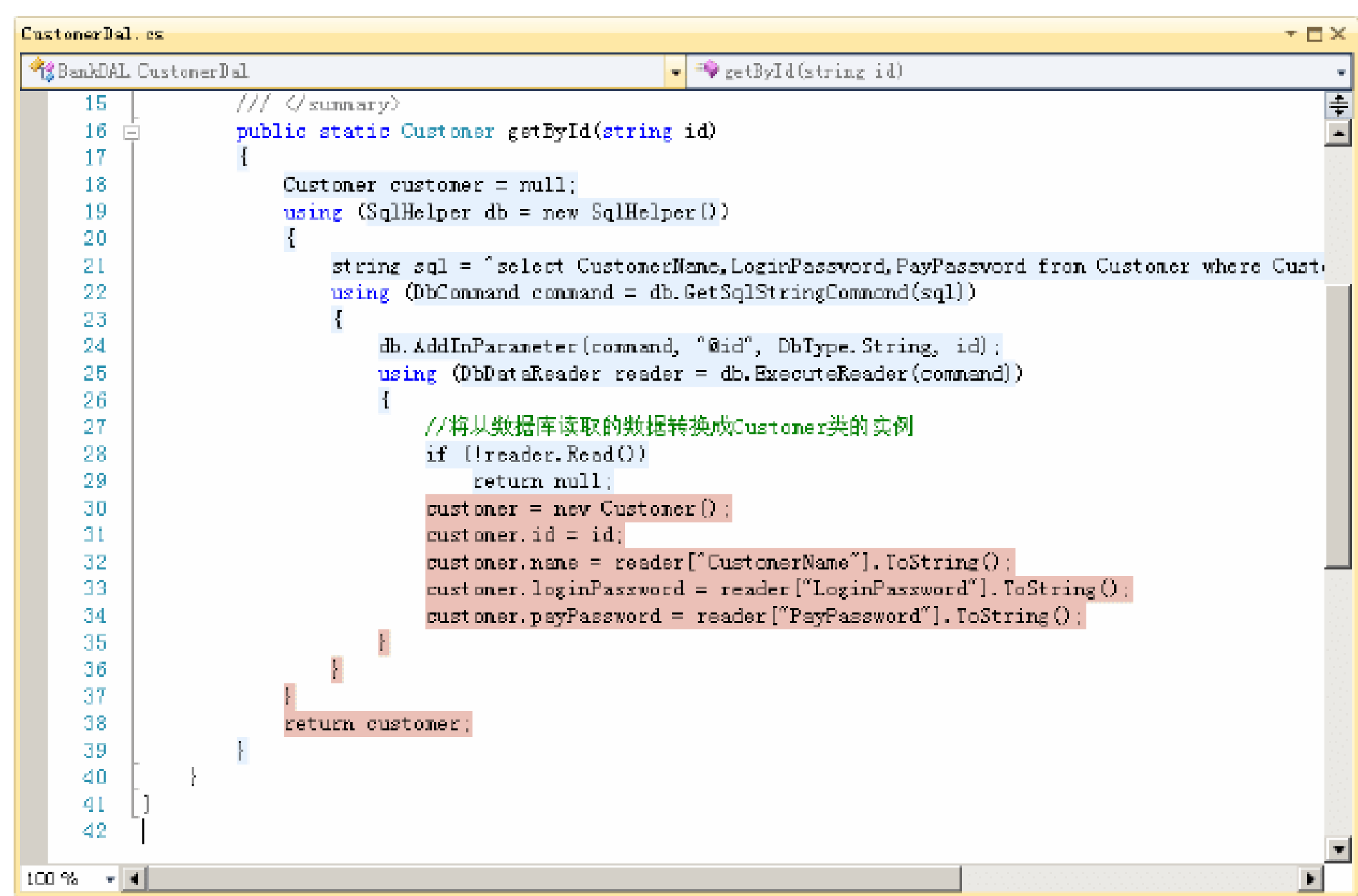


图 5.36　代码覆盖详情

5.5　Web 测试

Visual Studio 不但支持单元测试，还支持其他类型的测试，如 Web 测试、负载测试、数据库单元测试等。Web 测试可以模拟用户在浏览器中对 Web 应用程序的操作，从而测试 Web 应用程序的功能和性能。Web 测试通过录制回放的方式运行，即先录制用户在浏览器中的操作，然后再重复执行用户的操作。

当用户输入一个网址浏览网页时，会向 Web 服务器发送一个 HTTP Get 请求，服务器则响应此请求返回一个 HTML 文件，浏览器以可视化的方式显示出 HTML 文件，用户就看到了页面。如果用户在页面中输入数据，并且通过提交按钮将数据回发给服务器时，会产生一个 HTTP Post 请求。Web 测试的原理就是记录用户操作过程中产生的 HTTP 请求，然后以编程的方式向 Web 服务器模拟发送这些请求，并接收和验证服务器返回结果。下面以测试银行转账程序 BankApp 为例，说明 Web 测试的方法和步骤。

提示： 由于 Web 测试需要记录用户打开的 URL，并模拟用户操作向这些 URL 请求或者回发数据，因此要求用户使用的 URL 是固定的，而 Visual Studio 开发时使用的内置服务器会随时启动和停止，而且端口号是动态生成的，不是一个固定的 URL，因此 Web 测试时不宜使用 Visaul Studio 的内置服务器，而应该将 Web 项目发布到 IIS 中或者其他永久的 Web 服务器上。

（1）首先需要发布 BankApp 项目。在解决方案资源管理器中，右击 BankApp 项目名称，在弹出的快捷菜单中选择“发布”选项，则弹出如图 5.37 所示的“发布 Web”对话框。

（2）在其中，可以直接将 Web 应用发布到 Web 服务器上，也可以发布到本地文件系统上。本例将 BankApp 发布到本地。从图 5.37 所示对话框中间的“发布方法”下拉列表框中选择“文件系统”选项，则对话框如图 5.38 所示。

（3）在对话框中单击右下方的省略号按钮，则弹出如图 5.39 所示的“目标位置”对话框，在对话框左侧选择“本地 IIS”，然后单击右上角的创建新的 Web 应用按钮以创建 Web 应用，将新应用命名为 BankWeb。最后单击“打开”按钮，则图 5.39 所示的对话框关闭，回到图 5.38 所示对话框中。单击“发布”按钮，则 BankApp 项目会发布为指定的 IIS 项目，其地址为 http://localhost/BankWeb。

（4）接下来要基于 http://localhost/BankWeb 创建 Web 测试。在 Visual Studio 中，从菜单中选择“测试”|“新建测试”命令，则弹出“添加新测试”对话框。在对话框的测试模板列表中选择“Web 性能测试”，并从对话框最下方项目列表中选择“创建新的 C#测试项目”，并将新项目命名为 BankWebTest，如图 5.40 所示。

（5）单击“确定”按钮，则在解决方案中添加了一个新的测试项目 BankWebTest。在项目上右击，从弹出的快捷菜单中选择“添加”|“Web 测试”命令，则会自动打开一个浏览器窗口，窗口左侧有录制、停止录制等按钮，如图 5.41 所示。

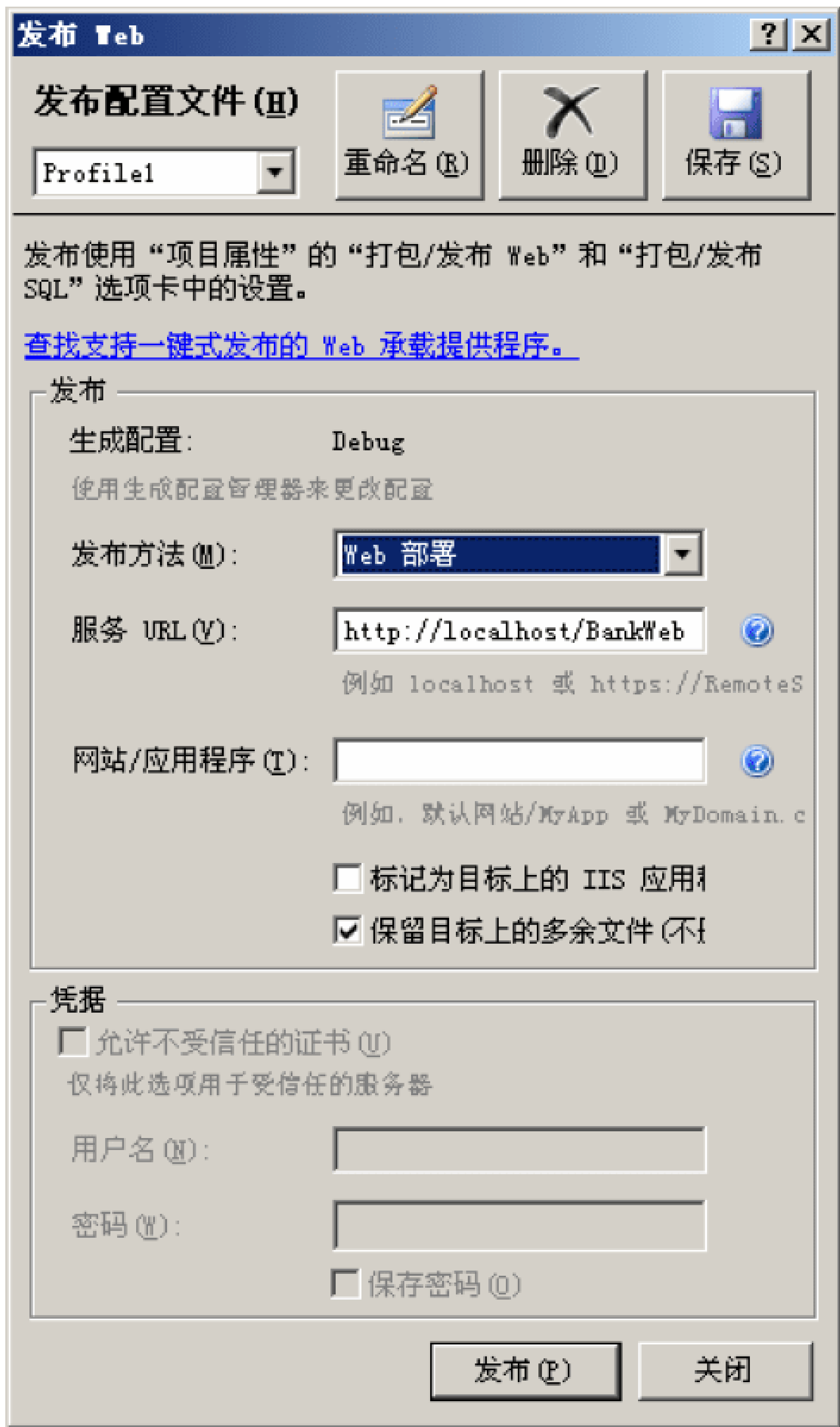


图 5.37　发布 Web 应用程序

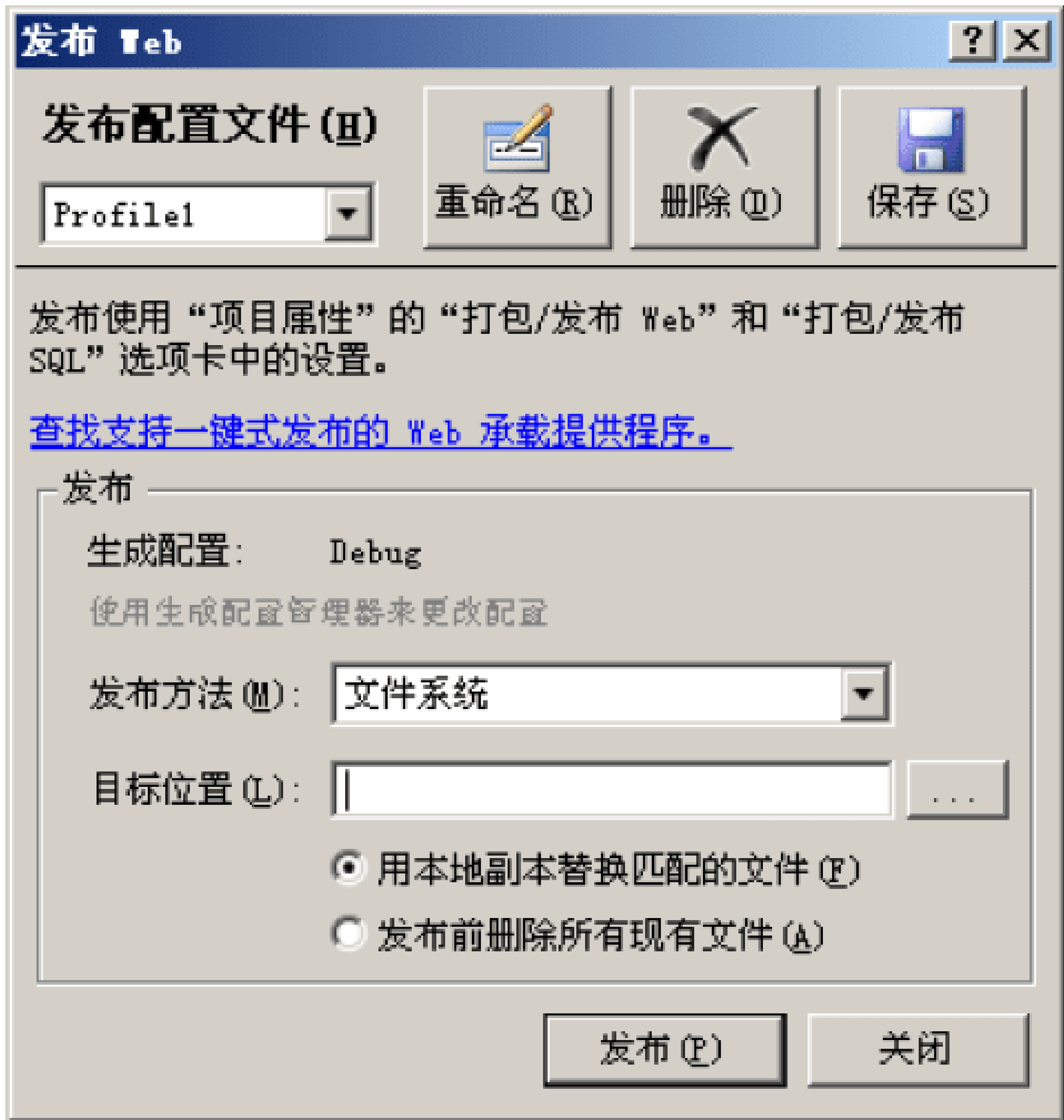


图 5.38　选择发布到文件系统

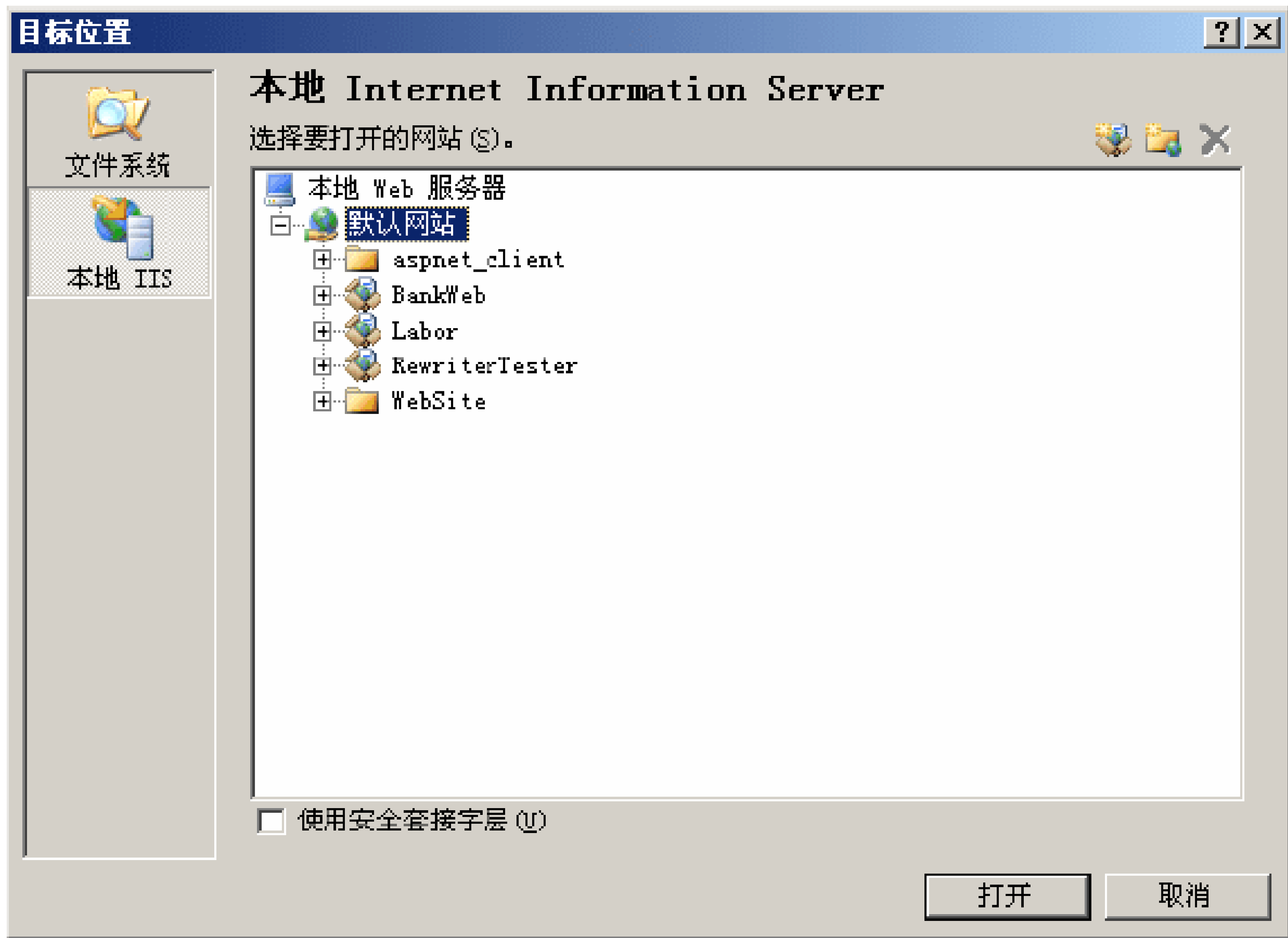


图 5.39　发布到本地 IIS

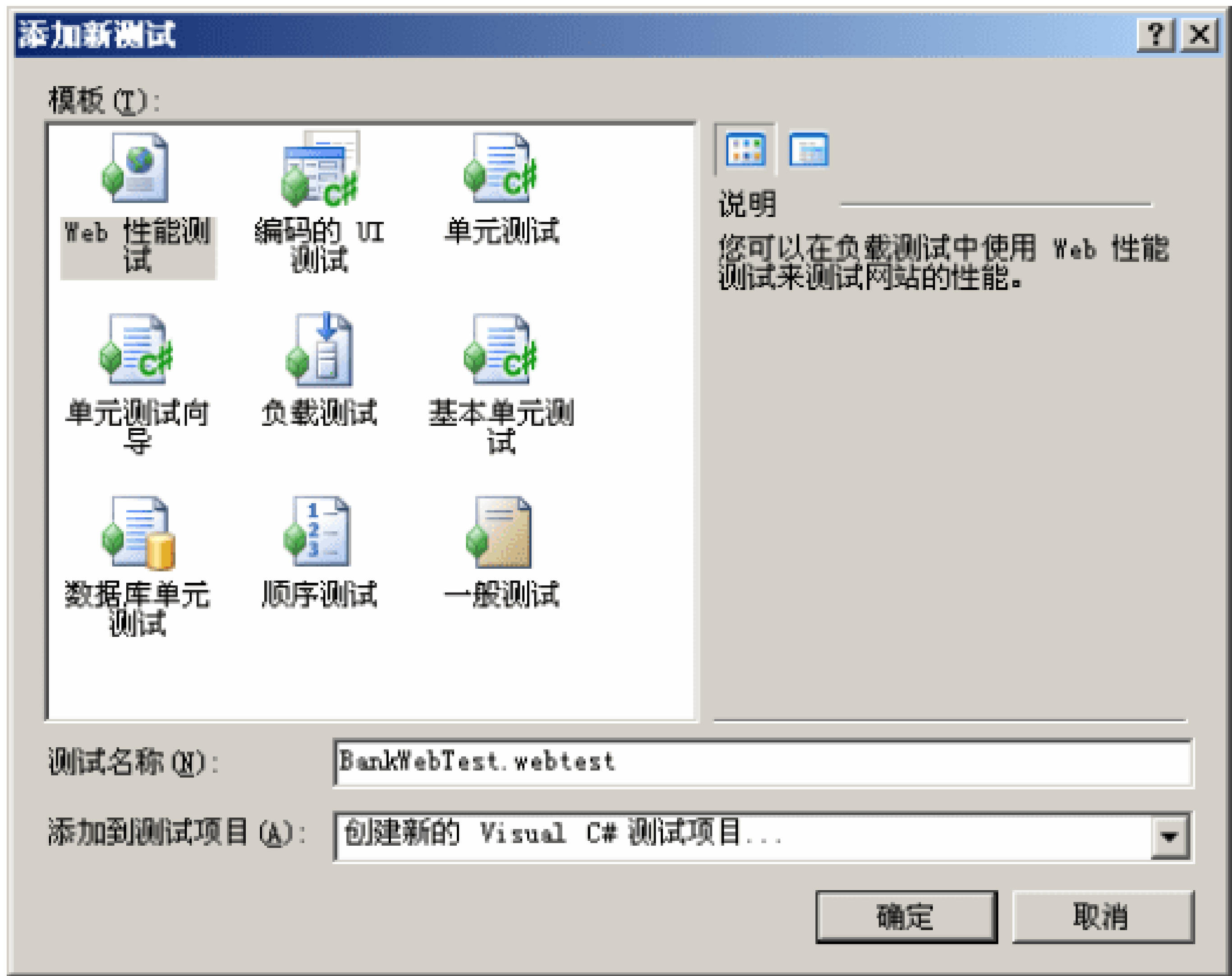


图 5.40　新建测试

（6）在图 5.41 所示的 Web 测试录制窗口中，启动录制后，用户在浏览器中所有操作都会被记录下来。为了测试银行转账程序，在浏览器地址栏中输入银行转账的 URL，可以看到，左侧的录制窗口中记录下了用户的操作，产生了新的数据，如图 5.42 所示。

（7）在图 5.42 所示窗口中执行其他需要测试的操作，此处为输入转账信息，然后单击转账按钮进行转账。可以先故意输入错误数据，然后再输入正确数据，加强对程序的测试。在浏览器中这些操作都被录制下来。

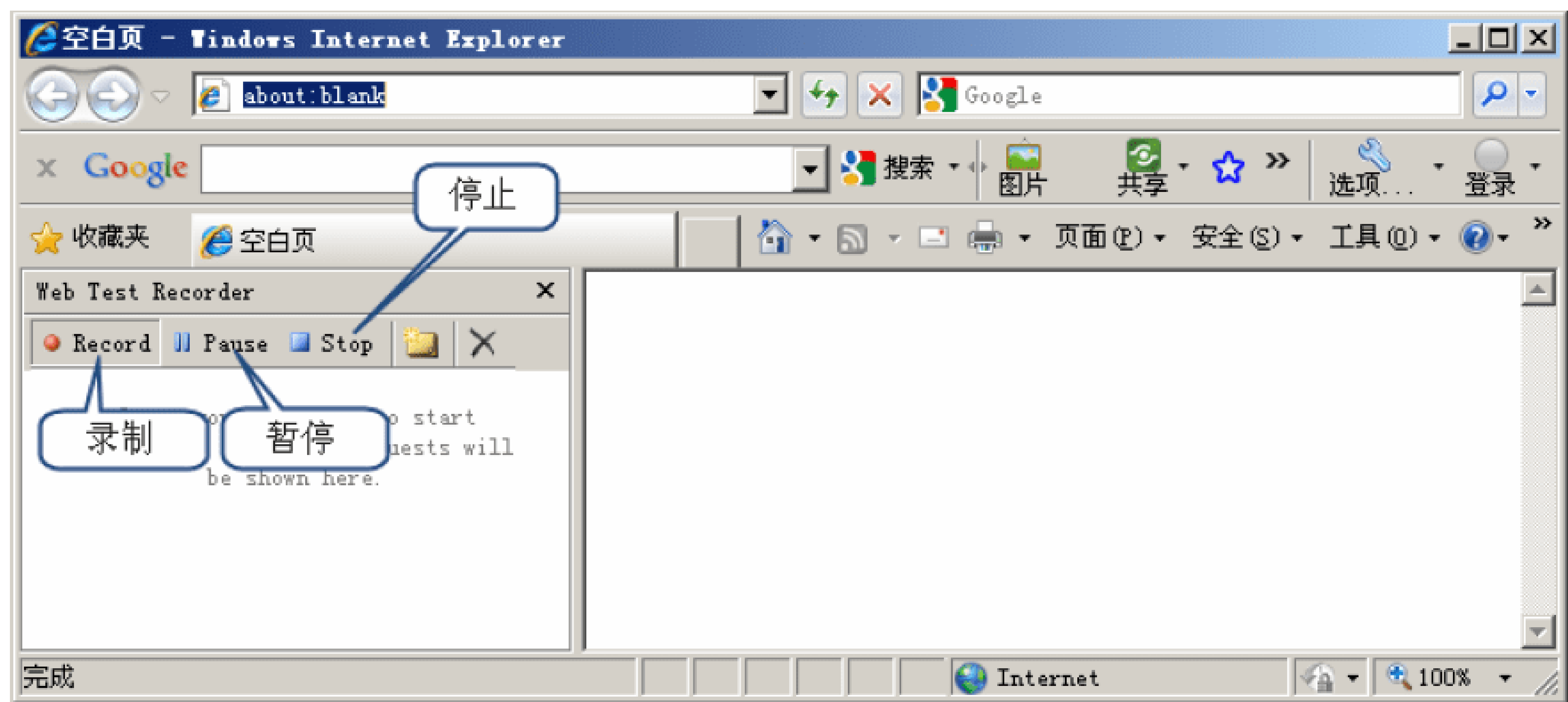


图 5.41　Web 测试录制窗口

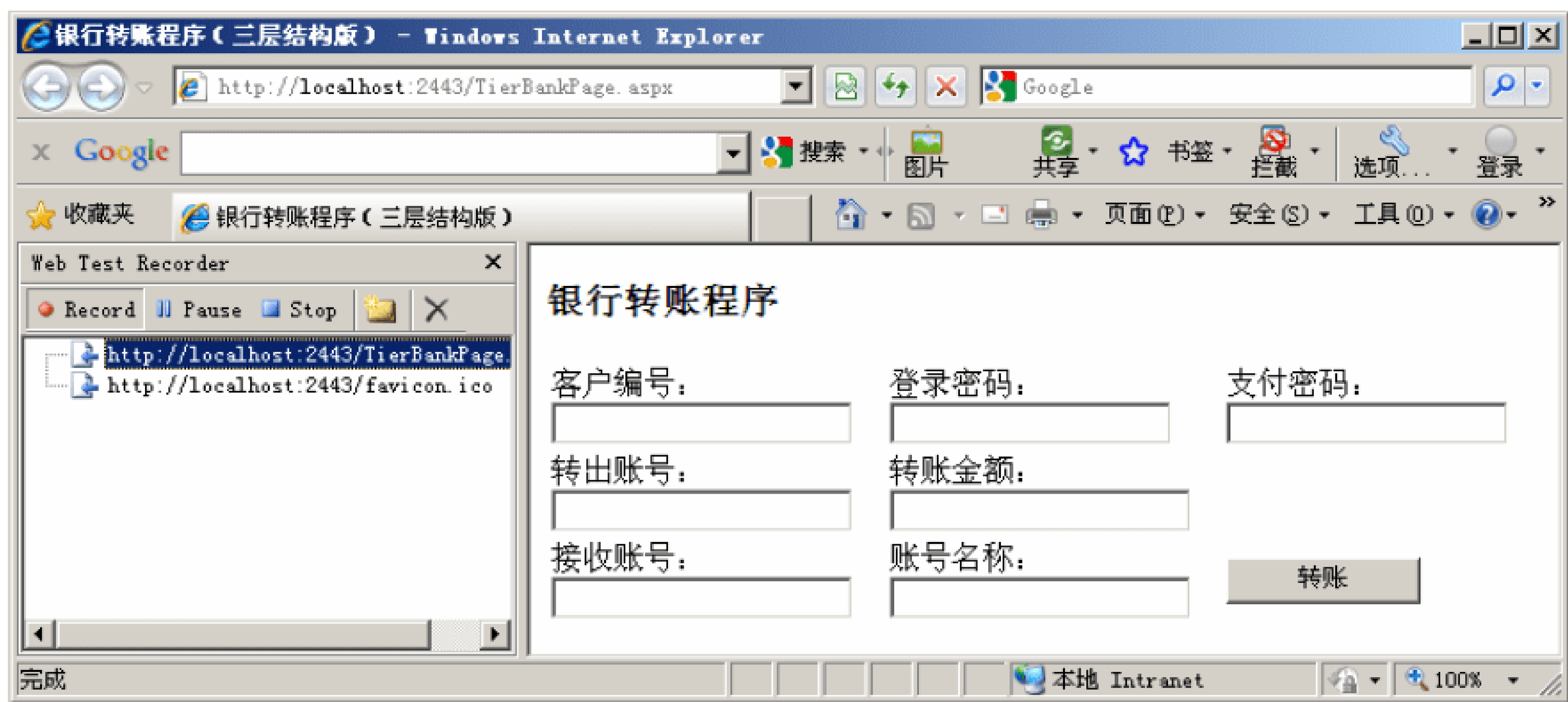


图 5.42　用户操作被录制

（8）录制完成后，单击“停止录制”按钮，则录制浏览器关闭，回到 Visual Studio 中，并且显示了刚才录制的脚本，如图 5.43 所示。

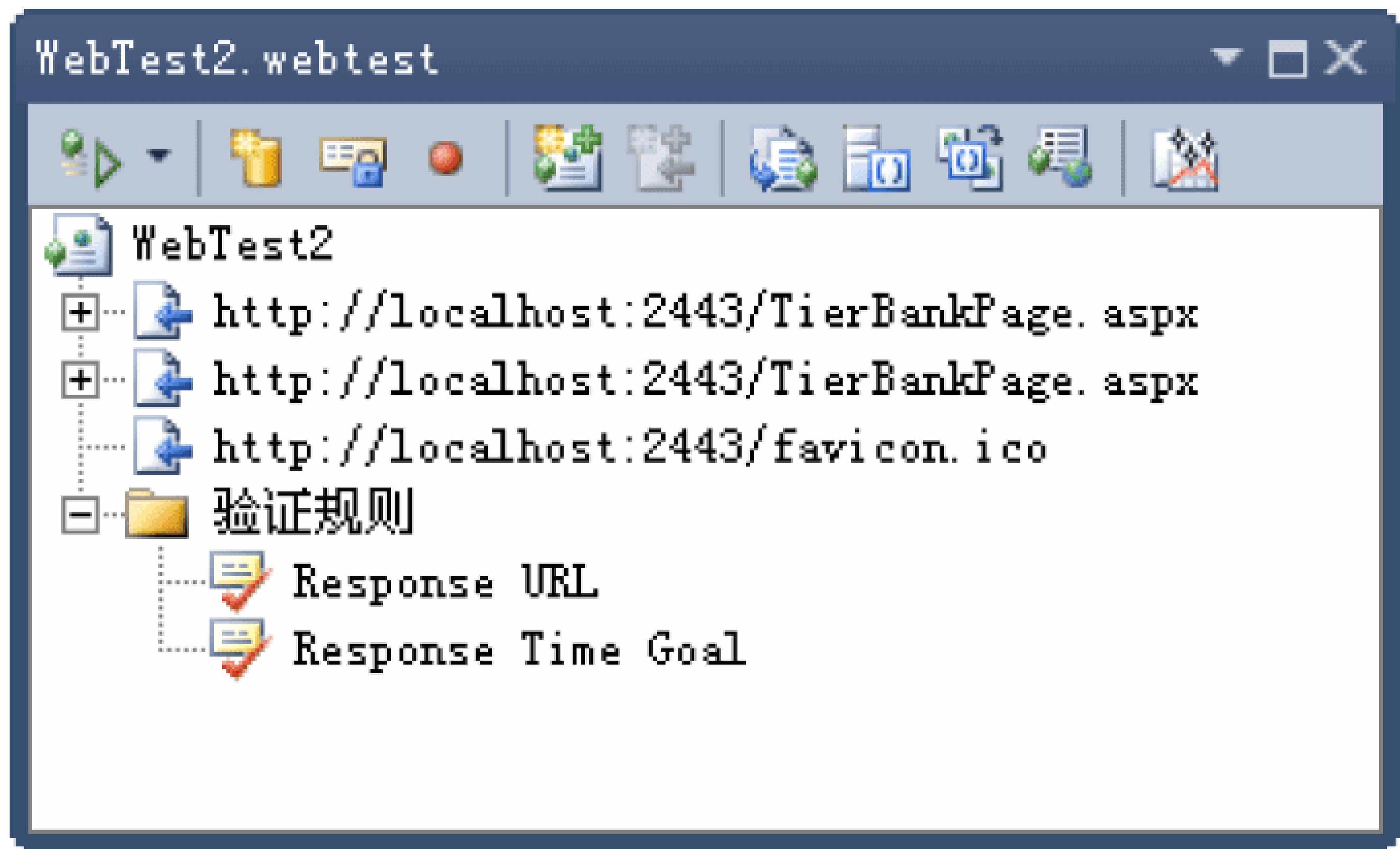


图 5.43　录制的脚本

（9）用户对录制的脚本进行编辑。如果要修改用户输入的目的账号，则在 Web 测试视图中展开单击“转账”按钮时所产生的 HTTP 请求，并从表单回发参数中选择目的账号所对应的表单字段 destination，即可在右侧属性窗口中修改此参数的值，如图 5.44 所示。

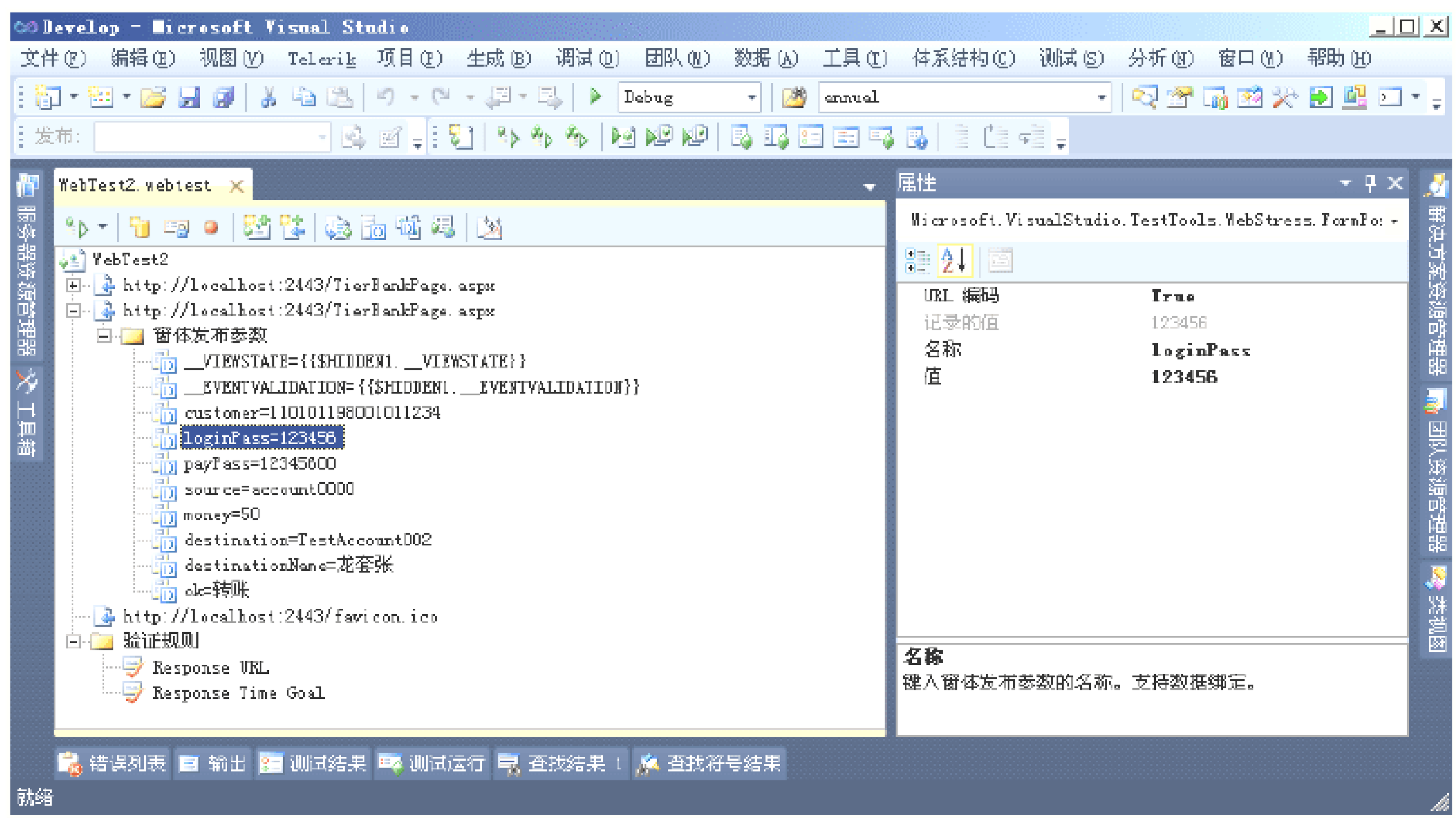


图 5.44　编辑录制的脚本

（10）当用户执行一个 Web 程序时，可以根据页面上显示的内容判断程序是否运行成功。在进行 Web 测试时，可以通过添加规则让测试程序根据页面上的内容进行判断。例如，在银行转账程序中，如果用户输入错误的账号，则页面上应该显示“顾客不拥有所请求账号”的错误提示。为 Web 测试添加验证规则的操作步骤为，在图 5.43 所示的 Web 测试视图中，右击“验证规则”文件夹，从弹出的快捷菜单中选择“添加验证规则”命令，则弹出如图 5.44 所示的对话框。

在图 5.45 所示的“添加验证规则”对话框左侧列出了一些常用验证规则，如验证某个 HTML 标记中的值、验证表单中某个字段、验证最大请求时间、验证页面上出现的文本等。在本例中，如果用户输入错误的账号，应该给出错误提示，所以从验证规则列表中选择“查找文本”，然后从右侧属性窗口中设置查找“顾客不拥有所请求账号”，并且设置如果找到文件则通过测试。

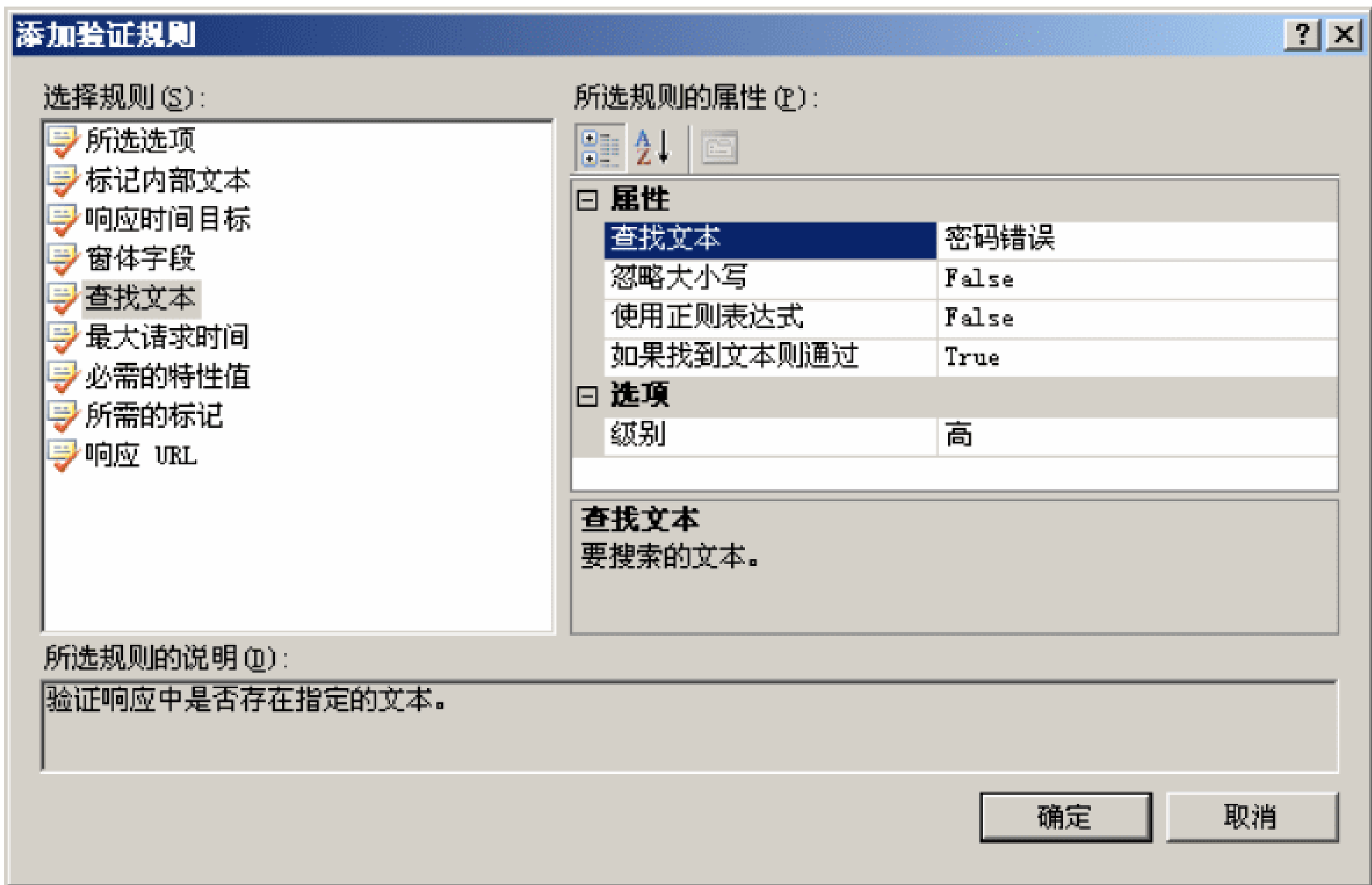


图 5.45　添加验证规则

（11）在测试视图中单击“运行测试”按钮，则 Visual Studio 会回放刚才所录制的用户操作。在 Visual Studio 窗口中显示了 Web 程序运行界面和每次 URL 请求执行情况，如是否成功、响应时间等，如图 5.46 所示。

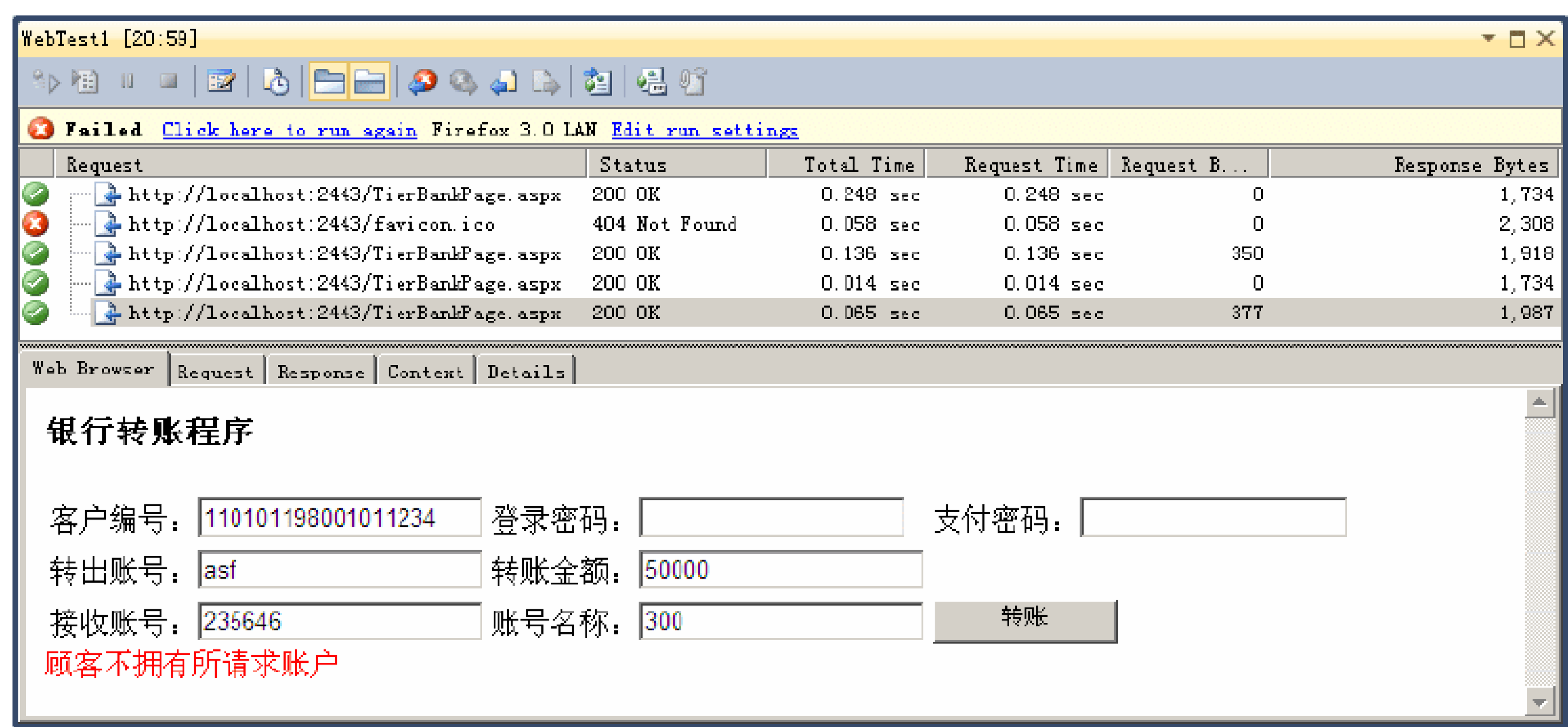


图 5.46　Web 测试运行界面

（12）如前所述，Web 测试的原理就是记录用户操作过程中产生的 HTTP 请求，然后以编程的方式向 Web 服务器模拟发送这些请求，并接收和验证服务器返回结果。这些步骤都对应于一定的代码，通过保存和手工编辑这些代码，用户可以实现更加个性化的需求。要查看 Web 测试所对应的代码，在 Web 测试视图中，右击 Web 测试名称，从弹出菜单中选择“生成代码”命令。本例所生成的关键代码如下。

```
//创建第一个 HTTP Get 请求，显示银行转账页面
WebTestRequest request1 = new WebTestRequest("http://localhost:2443
/TierBankPage.aspx");
request1.ThinkTime = 77;
ExtractHiddenFields extractionRule1 = new ExtractHiddenFields();
extractionRule1.Required = true;
extractionRule1.HtmlDecode = true;
extractionRule1.ContextParameterName = "1";
request1.ExtractValues += new EventHandler<ExtractionEventArgs>
(extractionRule1.Extract);
yield return request1;
request1 = null;
//创建第二个 HTTP 请求，为一个 Post 请求，向服务器提交转账信息
WebTestRequest request2 = new WebTestRequest("http://localhost:2443
/TierBankPage.aspx");
request2.Method = "POST";                                //设置为 POST 方法
FormPostHttpBody request2Body = new FormPostHttpBody();
//以下语句添加 Post 的各个参数
request2Body.FormPostParameters.Add("customer", "110101198001011234");
request2Body.FormPostParameters.Add("loginPass", "123456");
request2Body.FormPostParameters.Add("payPass", "12345600");
request2Body.FormPostParameters.Add("source", "account0000");
request2Body.FormPostParameters.Add("money", "50");
request2Body.FormPostParameters.Add("destination", "TestAccount002");
request2Body.FormPostParameters.Add("destinationName", "龙套张");
request2Body.FormPostParameters.Add("ok", "转账");
request2.Body = request2Body;
yield return request2;
request2 = null;
```

5.6　负 载 测 试

在进行软件测试时，大多数情况下都需要测试软件在多用户并发环境下的运行情况，包括性能、响应时间、稳定性等。Visual Studio 中的负载测试可以模拟多用户并发执行的环境，满足以上测试要求。负载测试由一系列 Web 测试或单元测试组成，这些测试在存在多个模拟用户的情况下运行一段时间。负载测试可以用于以下几种不同的测试类型。

- 冒烟测试：确定在短时间内负载较小时应用程序如何执行。
- 压力测试：确定在较长时间内负载较大时应用程序是否能成功运行。
- 性能测试：确定应用程序的响应能力。
- 容量计划测试：确定在各种容量下应用程序如何执行。

下面以银行转账程序测试为例，说明 Visual Studio 中进行负载测试的步骤。

（1）打开银行转账程序，从菜单中选择“测试”|“新建测试”命令，在打开的“新建测试”对话框中，选择“负载测试”，则会显示“新建负载测试向导”欢迎对话框。在其中单击“下一步”按钮，进入“负载方案设置”对话框，在其中可以设置方案名称和用户思考时间，如图 5.47 所示。本例的思考时间采用以记录的思考时间为中心的正态分布。

提示： 思考时间用于模拟人类与网站执行的各种交互之间存在等待时间这种行为。Web 测试中的各个请求之间及负载测试方案的各个测试迭代之间均会产生思考时间。在负载测试中使用思考时间对于创建更为精确的负载模拟很有用。

（2）“负载测试向导”的下一步骤为设置负载模式，可以选择固定的用户数或者分级负载。本例选择固定 50 个用户数，如图 5.48 所示。

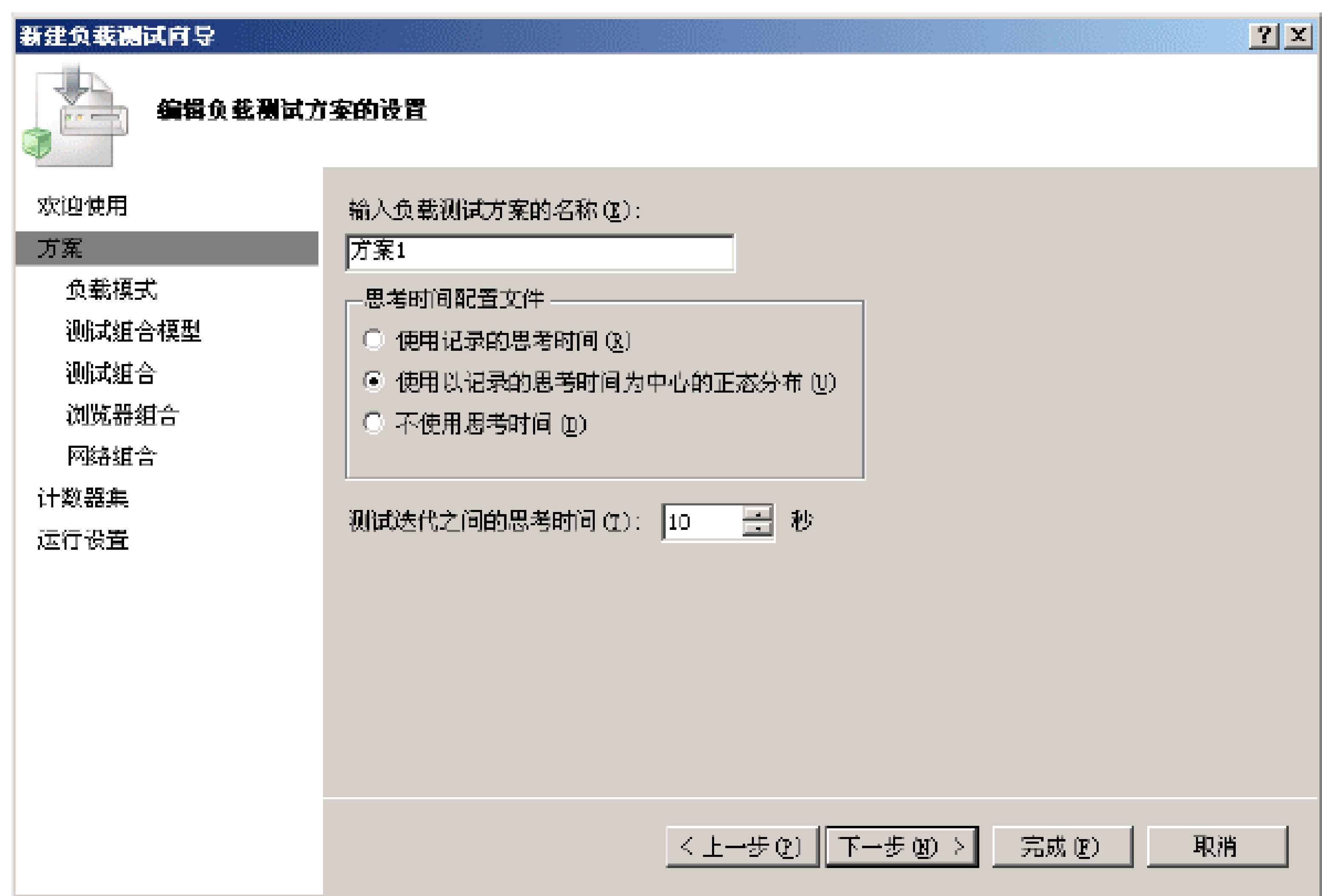


图 5.47　设置测试方案

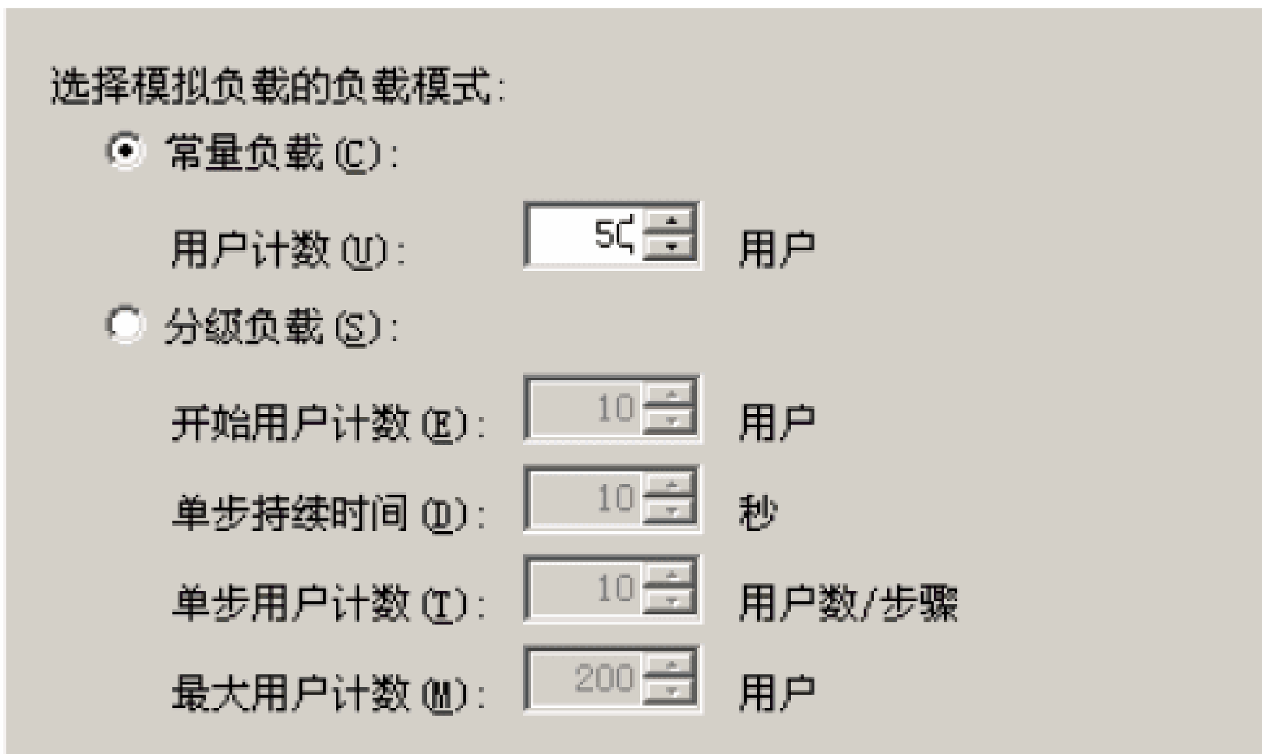


图 5.48　设置负载模式

（3）在“负载测试向导”中单击“下一步”按钮，直到进入如图 5.49 所示的测试组合步骤，在此可以指定负载测试中包含的单元测试和 Web 测试。

（4）在图 5.49 所示对话框中单击“添加”按钮，则打开如图 5.50 所示的“添加测试”对话框。对话框左侧列出了当前项目中存在的测试，右侧列出了包含在负载测试中的测试。从左侧选择需要添加的测试，单击对话框中间的“>”按钮，将其添加到负载测试中。



图 5.49　负载测试组合

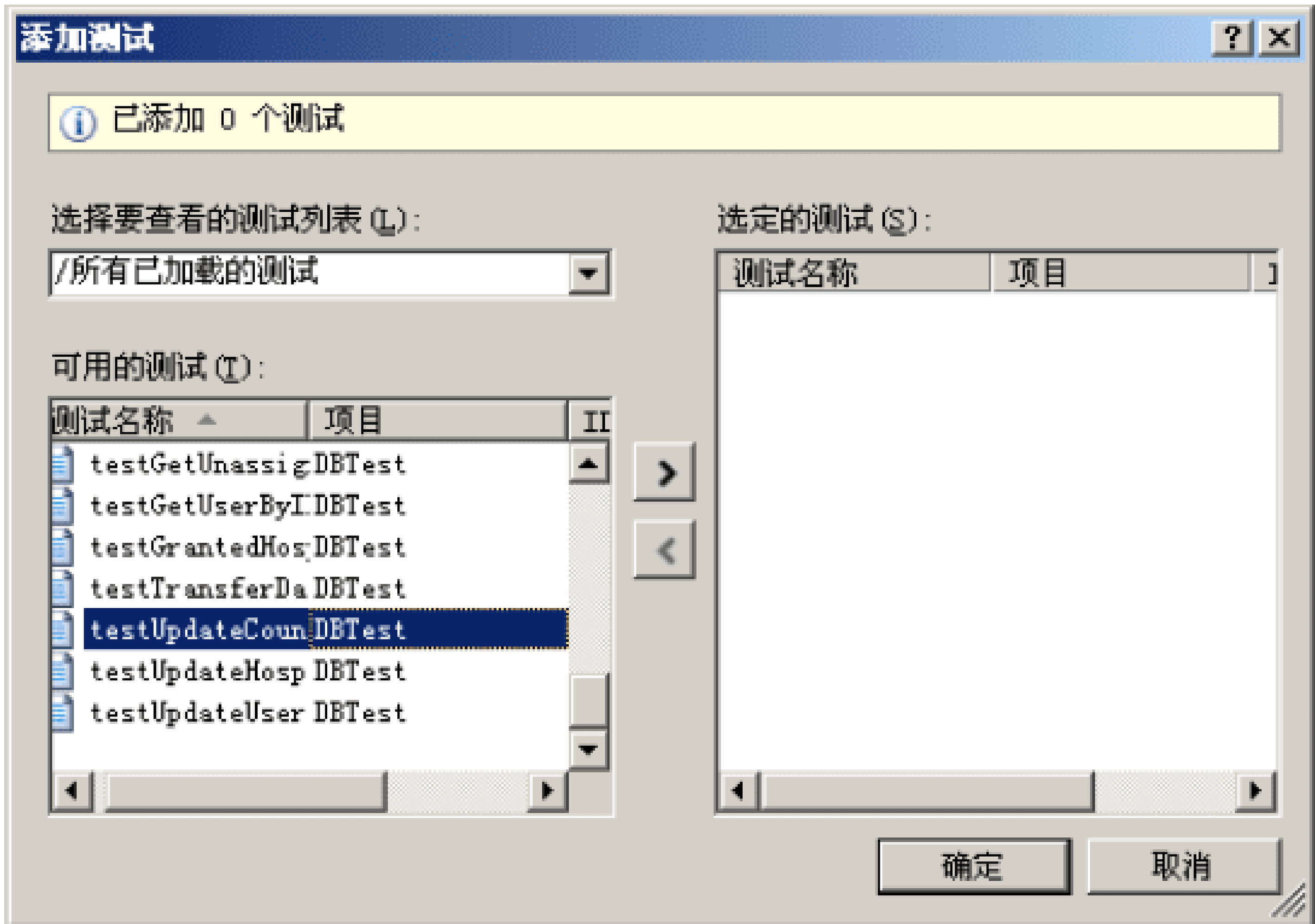


图 5.50　向负载测试中添加测试

（5）在“负载测试向导”的下一步骤中，可以指定测试所处的模拟网络环境。可以选择局域网或者不同速度的广域网的组合为模拟网络环境，如图 5.51 所示。

（6）在“负载测试向导”的下一步骤中，可以为测试指定不同的浏览器组合，例如，可以使用 FireFox、Internet Explorer、Netscape 等不同的浏览器对银行转账程序进行测试，如图 5.52 所示。

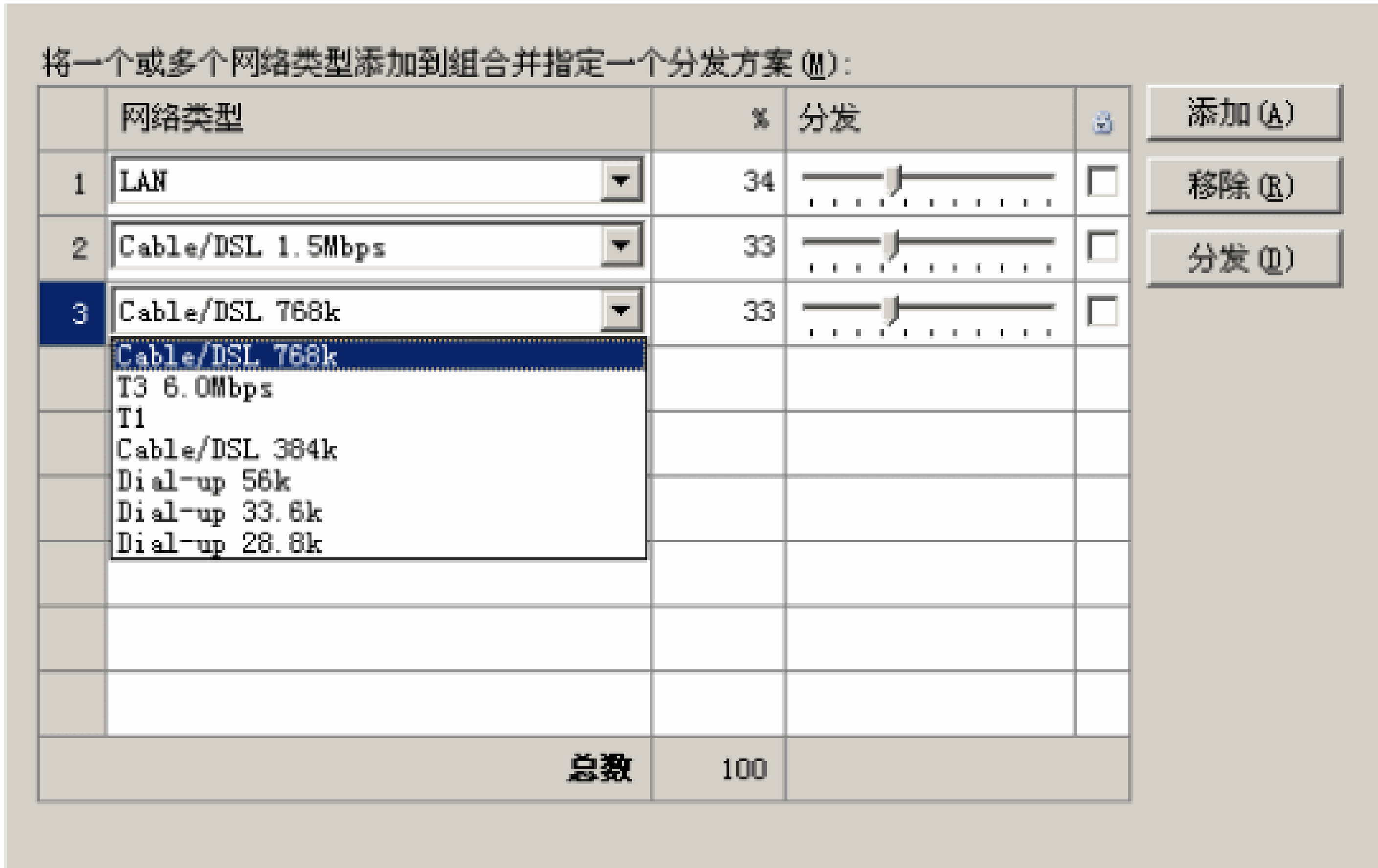


图 5.51　配置模拟网络环境



图 5.52　浏览器组合

（7）在“负载测试向导”最后一个步骤中，可以指定测试持续运行的时间或者重复执行的次数，如图 5.53 所示。

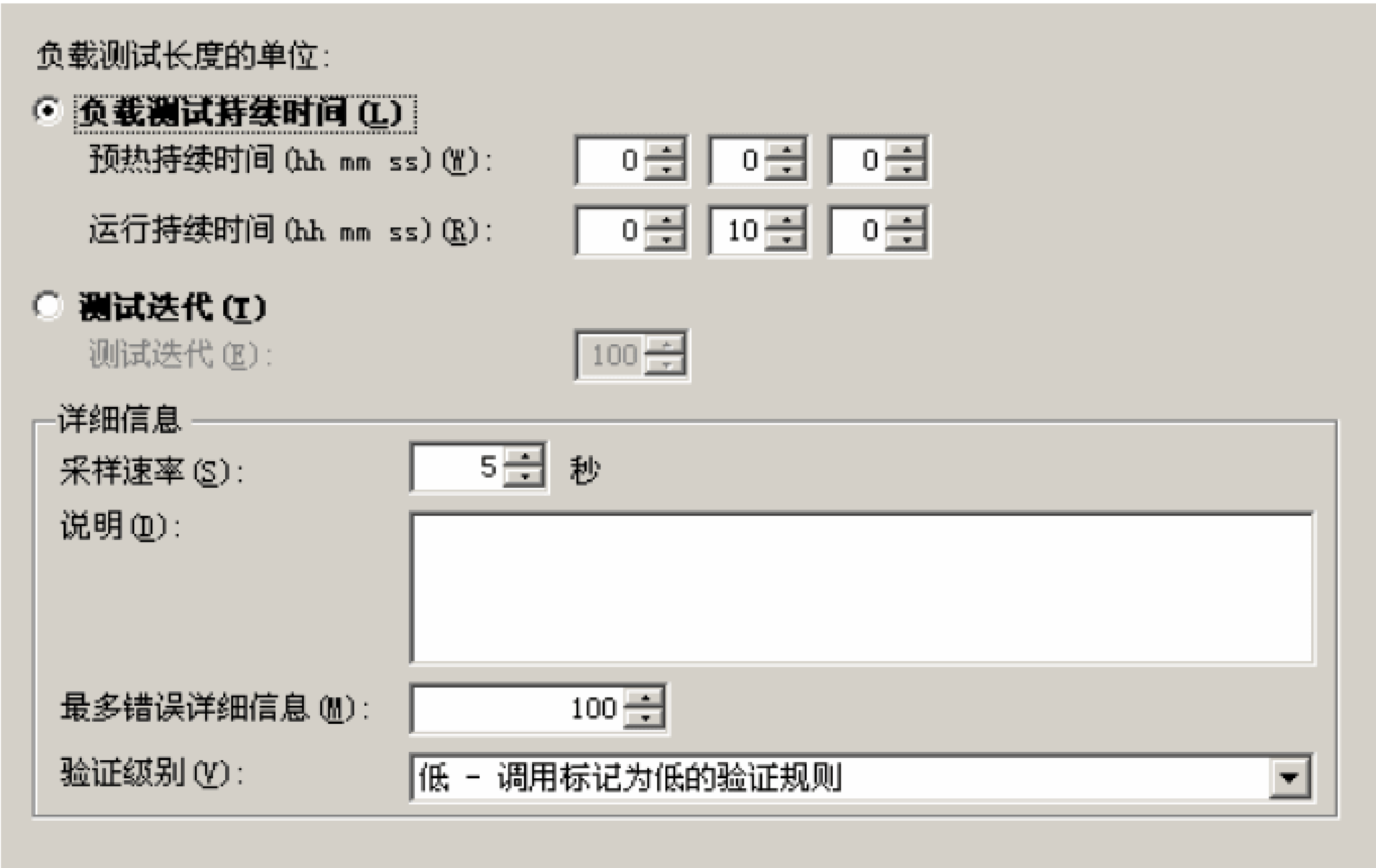


图 5.53　负载测试运行设置

（8）负载测试创建完成后，单击工具栏上的运行测试按钮以运行该测试。测试运行过程中，将以图表方式实时显示各个数据指标，如图 5.54 所示。

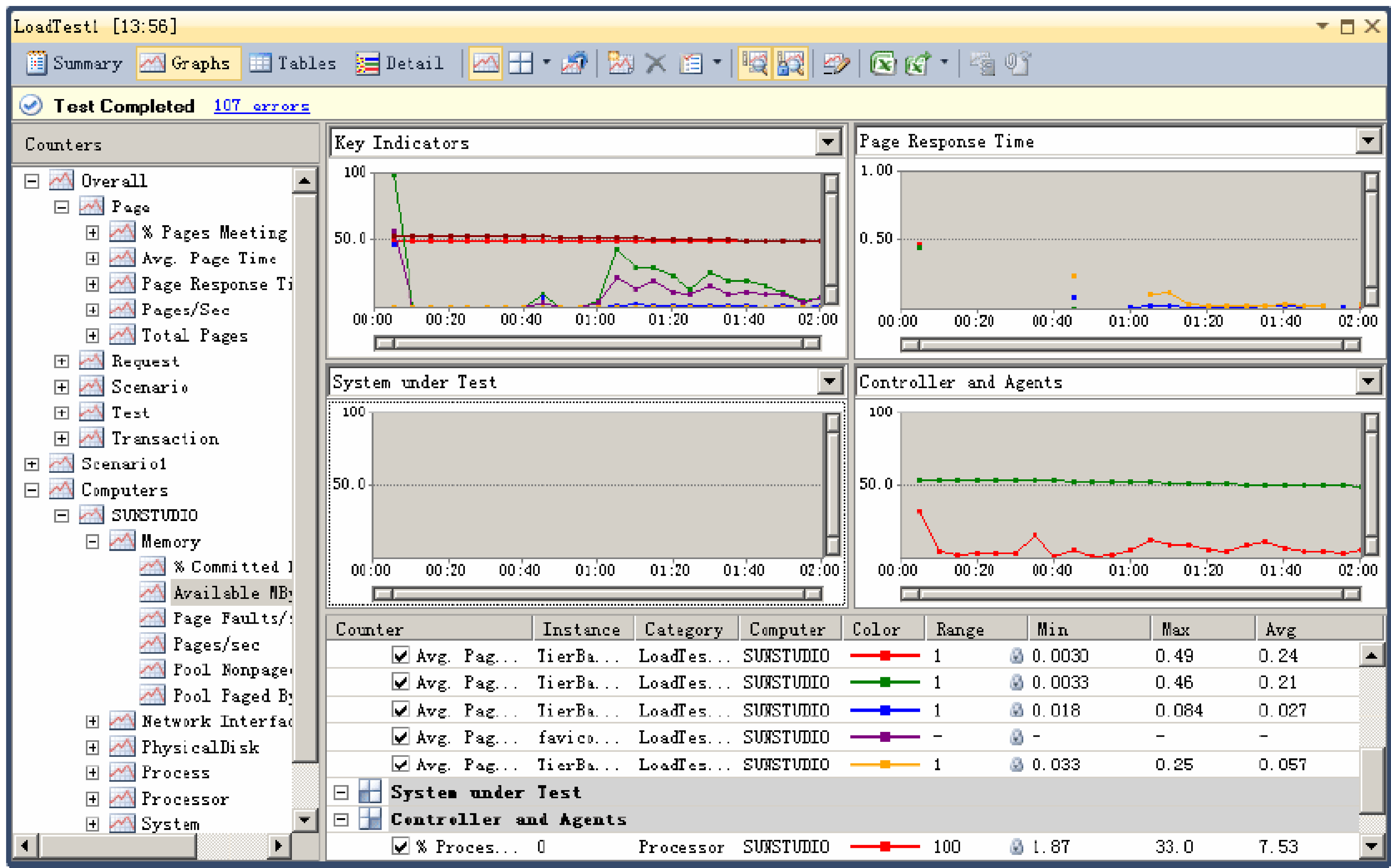


图 5.54　负载测试运行监控

5.7　小　　结

本章主要介绍了 3 部分内容：源码管理、三层结构和软件测试。首先介绍了 Visual SourceSafe 源码管理工具的使用及其与 Visual Studio 的集成，接下来介绍了三层结构并分析其优势，最后介绍了几种常用的软件测试：单元测试、Web 测试和负载测试。

第 6 章　搜索引擎优化

现在互联网已经成为人们日常生活中一个重要的组成部分，人们上网看新闻，在网上逛商店，在论坛中讨论问题等。广大企业和商家也充分意识到了互联网平台的重要性，纷纷建立企业网站，发布供求信息。企业网站建立以后，如何让更多的用户来访问是一个非常重要的问题。

在当今互联网上，用户查找某种资料的方式一般都是使用搜索引擎进行搜索，然后根据搜索引擎列出的搜索结果和链接访问相关网站。这就意味着，在搜索引擎的搜索结果中，排名越靠前的网站被访问到的机率越高，而更高的访问次数也就意味着更多的商品销售和利润。如何才能使自己的网站在搜索引擎的搜索结果中排名靠前呢？这对企业来说是一个很重要的问题，这也就是搜索引擎优化所要解决的问题。

6.1　搜索引擎优化简介

搜索引擎优化对应的英文名称是 Search Engine Optimization，简称 SEO，是近年来较为热门的一个技术领域。本节将介绍搜索引擎优化的概念、动机和原理，使读者对搜索引擎优化建立一个印象。

6.1.1　搜索引擎优化基本概念

随着互联网的普及，越来越多的企业和商家建立自己的网站，在互联网上发布自己的产品信息，甚至有很多个人也建立了个人网站或者博客。作为一个展示产品的平台和媒介，相对于传统的产品宣传方式来说，互联网拥有更丰富的内容、更广泛的受众和更廉价的成本，因而也越来越得到企业和商家的重视。

对于一个企业来说，为了达到宣传企业商品的目的，建立企业网站只是第一步。网站建立后，如果没有顾客访问，则起不到任何作用。企业要想提高网站的访问量，可以有两种主要途径，第一是为网站做广告，这种方式成本较高；第二就是搜索引擎优化，这种方式成本较低且效果明显。

所谓搜索引擎优化，是指通过总结搜索引擎的排名规律，对网站进行合理优化，使目标网站在搜索引擎（如 Google 和百度）的排名提高，从而获得更多的用户访问量。搜索引擎优化技术的出现与搜索引擎的广泛应用密不可分。现在人们要从网上查找某些信息时，通常都会通过搜索引擎针对某关键字进行搜索，然后再从搜索引擎的搜索结果中点击链接进入相应网站。

由于互联网的海量信息量，对于一个关键字的搜索通常会有庞大的搜索结果，搜索引

擎以分页的方式显示这些结果。显然列在第一页的搜索结果会有最大的机率得到访问，而排列在第几十、几百、甚至几千页以后的搜索结果几乎不会被访问。例如，在 Google 中搜索“烤箱”，则得到获得约 4 830 000 条结果，若每页显示 10 条结果，则一共有 48 万页。对于一个生产烤箱的企业来说，当然希望自己厂家的网站能够显示在搜索结果第一页甚至第一条的位置，而不希望在位于搜索结果的几十页以后。

如何才能使自己的网站在搜索结果中排名靠前呢？这就需要理解和掌握搜索引擎的工作原理和对搜索结果进行排名的算法，并且以此为根据有针对性的完善自己的网站，从而提高自己网站的搜索引擎排名。

6.1.2　搜索引擎工作原理

搜索引擎按其工作方式主要分为 3 种，分别是全文搜索引擎（Full Text Search Engine）、目录索引类搜索引擎（Search Index/Directory）和元搜索引擎（Meta Search Engine）。

全文搜索引擎从互联网上提出各个网站和页面信息，并把这些信息处理后保存到数据库中，用户使用搜索引擎进行搜索时，从数据库中检索与用户查询条件匹配的记录，并按照一定的条件对查询结果进行排序后将结果返回给用户。全文搜索引擎是真正意义上的搜索引擎，比较典型的有国外的 Google 和国内的百度。本章所讲的搜索引擎优化主要针对此类搜索引擎。

目录索引虽然有搜索功能，但在严格意义上讲算不上是真正的搜索引擎，仅仅是按目录分类的网站链接列表而已。目录搜索引擎收录了互联网上存在的网站，给每个网站分配若干个标签，用户可以根据标签找到相关网站。例如，用户可以通过“编程”、“游戏”、“新闻”等标签找到相关的网站。比较典型的目录索引有雅虎、搜狐等。

元搜索引擎是建立在其他搜索引擎基础上的搜索引擎。当用户在元搜索引擎中进行查询时，元搜索引擎把查询条件传递给其他多个搜索引擎，使用其他搜索引擎进行搜索，然后将查询结果返回给用户。比较典型的元搜索引擎有 MetaSearch（http://metasearch.com/）、觅搜（http://www.metasoo.com/）等。

鉴于目前主流的搜索引擎为全文搜索引擎，本章将以这种搜索引擎为例说明搜索引擎工作原理和优化技术。当用户使用搜索引擎进行搜索时，并不是实时在互联网上进行搜索，而是在一个预先整理好的网页索引数据库中进行搜索，然后将搜索结果排序并显示给用户。搜索引擎的工作原理包含 3 个步骤：从互联网上抓取网页；建立索引数据库；在索引数据库中搜索排序。

1. 抓取网页

页面抓取是指从互联网上不断访问页面并取回页面内容的过程，这一工作是搜索引擎最基础的工作。互联网上的页面是由 HTML 文件组成的，这些 HTML 文件相互链接，形成一个网状结构。搜索引擎的页面抓取程序通过这些链接从一个页面到达另外一个页面，好像一个蜘蛛在页面组成的网上爬来爬去，因此搜索引擎的页面抓取程序通常被形象的称为“蜘蛛（Spider）”或者“爬虫（Crawler）”。

2. 建立索引数据库

搜索引擎蜘蛛把网页数据抓取回来以后，需要对这些内容进行分析整理，提取相关网页信息（包括网页所在 URL、编码类型、页面内容包含的关键词、关键词位置、生成时间、与其他网页的链接关系等），根据一定的相关度算法进行大量复杂计算，得到每一个网页针对页面内容中及超链中每一个关键词的相关度（或重要性），然后用这些相关信息保存到一个特定结构的索引数据库中。

3. 搜索和排序

当用户输入关键词搜索后，由搜索系统程序从网页索引数据库中找到符合该关键词的所有相关网页。因为所有相关网页针对该关键词的相关度早已算好，所以只需按照现成的相关度数值排序，相关度越高，排名越靠前。最后，由页面生成系统将搜索结果的链接地址和页面内容摘要等内容组织起来返回给用户。

搜索引擎蜘蛛一般要定期重新访问所有网页（各搜索引擎的周期不同，可能是几天、几周或几个月，也可能对不同重要性的网页有不同的更新频率），更新网页索引数据库，以反映出网页内容的更新情况，增加新的网页信息，去除死链接，并根据网页内容和链接关系的变化重新排序。这样，网页的具体内容和变化情况就会反映到用户查询的结果中。

6.1.3　搜索引擎排名因素

搜索引擎排名算法非常复杂，要综合考虑许多因素形成对一个页面的评价。本节将介绍影响搜索引擎排名的主要因素，搜索引擎优化就是针对这些因素完善页面，使搜索引擎对页面的评价增加，从而提高在搜索结果中的排名。影响一个页面在搜索引擎中排名的因素可以分为两大类：页面内部因素和页面外部因素。页面内部因素包括以下内容：

1. 页面标题

页面标题即 HTML 文档中<title>标记中的内容。页面标题应该明确说明页面的内容，最好包含搜索关键字。给页面指定一个合适的标题能够大大提高搜索引擎对页面的评价值。页面标题应该与页面内容密切相关，不能使用太笼统的标题，更不能使标题与内容无关。一个网站中要避免所有页面具有完全相同的标题。

2. 正文标题

此处使用正文标题是为了与页面标题相区别。页面标题是指 HTML 文档中<title>标记中的内容，在浏览器中显示于浏览器窗口标题栏。而正文标题是指 HTML 文档中<h1>、<h2>等标记内容。通常这些标题说明了页面的主要内容和大纲，对搜索引擎来说，这些标题具有比正文更大的权重。如果页面的正文标题中包含与搜索关键字密切相关的内容，则搜索引擎对此页面的评价更高。

3. 页面内容

页面正文中应该包含与被搜索关键字相关的文字和图片。页面中出现搜索关键字的密

度越高，则说明页面与被搜索内容相关性越高，从而具有更好的搜索引擎评价。对于页面中的图片要设置其替换文本（即 img 标记中的 alt 属性）。替换文本的作用主要有两个，一方面是当图片无法加载时可以显示替换文本作为说明性文字，另一方面是由于搜索引擎目前无法识别图片内容，只能根据替换文本来判断图片内容。

影响一个页面搜索引擎排名的外部因素主要包括到此页面的外部链接，包括链接的数量、相关性、链接文本等。一个页面被越多外部链接引用，就说明这个页面越重要，搜索引擎对其评价也就越高。链接数量只是一个因素，不同的链接其质量也不同。例如，一个来自著名网站的链接和一个来自不知名网站的链接其价值是不一样的。对于外部链接还应考虑其相关性，例如，如果页面 a 是一个讲编程知识的页面，那么在链接到页面 a 的所有链接中，如果大部分都来自编程相关的页面，则认为这些链接质量较好，而来自于其他无关行业页面的链接质量就不好。

综上所述，如果一个页面具有明确的与内容相关的标题、页面内容与内容密切相关、页面拥有高质量的外部链接，那么这个页面在搜索引擎中就具有较好的排名。页面标题、内容、外部链接也是搜索引擎优化的主要方向，通过合理的设置页面标题和内容，想办法让其他页面引用自己的页面，就可以提高自己页面的搜索引擎排名。

6.1.4　SEO 作弊

如果所述，通过设置合适页面标题、页面文本、图片替换文本、页面链接等，可以提高页面的搜索引擎排名。但是如果滥用这些因素，甚至做假试图欺骗搜索引擎，不但达不到预想的目的，而且还会受到搜索引擎的惩罚，被降低排名甚至从搜索引擎结果中除名。在做搜索引擎优化时，需要了解这些作弊手段，以避免无意中使用受到搜索引擎惩罚。搜索引擎优化常用作弊手段主要有以下几种。

1. 关键字堆砌

为了增加关键词的出现频次，故意在网页代码中，如在 meta、title、注释、图片 alt 等地方重复书写某关键词的行为。下面的页面代码是一个关键字堆砌的例子，此页面为了提高 ASP.NET、C#、案例、教程等关键字搜索排名，在页面中添加了许多关键字，而这些关键字仅仅是堆砌而已，不是出现在正文的句子中，这是关键字堆砌作弊。

```
<head>
    <title>ASP.NET 案例教程</title>
</head>
<body>
<!--以下为关键字堆砌-->
ASP.NET C# jQuery Web Service SQL Server 编程 项目 案例 教程
ASP.NET C# jQuery Web Service SQL Server 编程 项目 案例 教程
ASP.NET C# jQuery Web Service SQL Server 编程 项目 案例 教程
ASP.NET C# jQuery Web Service SQL Server 编程 项目 案例 教程
ASP.NET C# jQuery Web Service SQL Server 编程 项目 案例 教程
<!--以下才是页面的真正内容（省略）-->
</body>
```

2. 虚假关键词

通过在 meta 或 title 中设置与网站内容无关的关键词，如设置热门关键词，以达到误导用户进入网站的目的。同样的情况也包括链接关键词与实际内容不符的情况。

HTML 文档中的<meta>元素可提供有关页面的元信息（meta-information），比如针对搜索引擎和更新频度的描述和关键词。meta 标签必须位于文档的头部，不包含任何内容。meta 标签的属性定义了与文档相关联的名称/值对，meta 标签与搜索引擎相关的属性主要有以下两个。

```
<meta name="keywords" content="搜索引擎从此处获得页面的关键词"/>
<meta name="description" content="搜索引擎从此处获得页面的主要内容"/>
```

Meta 元素中的内容对于用户来说是不可见的，但是却可以被搜索引擎访问到。如上代码所示，通过在 meta 标记中添加适当内容，可以通知搜索引擎页面此页面的关键词和主要内容。如果在 meta 中指定了某些热门关键词或内容，而页面上的实际内容与此无关，这种现象就是虚假关键词作弊。以下是一个虚假关键词的例子，例如在电影阿凡达刚上映时，网上许多人都在搜索阿凡达，为了提高页面被搜中的概率，在页面的<meta>中添加了虚假的关键词，试图欺骗搜索引擎。

```
<head>
   <meta name="keywords" content="阿凡达, 高清, 电影"/>
   <meta name="description" content="电影阿凡达高清在线观看"/>
</head>
<body>
页面中的真实内容，其实是一些其他文字，与阿凡达无关，更不能在线观看
</body>
```

3. 垃圾链接

“链接工厂”（也称为“大量链接机制”）指由大量网页交叉链接而构成的一个网络系统。一个站点加入“链接工厂”后，一方面可得到来自该系统中所有网页的链接，同时作为交换需要奉献自己的链接，籍此方法来提升链接得分。

4. 隐形文本和链接

为了增加关键词的出现频次，故意在网页中放一段与背景颜色相同的、包含密集关键字的文本。这样，访问网页的用户看不到，搜索引擎却能找到。类似方法还包括超小号文字、文字隐藏层等手段。隐形链接是在隐形文本的基础上在其他页面添加指向目标优化页的行为。以下是一个隐形文本作弊的例子。

```
<head>
   <title>ASP.NET 案例教程</title>
   <style type="text/css">
   /*将前景色背景色都设置为白色以达到隐藏目的*/
   .hiddenText1 { background-color:White; color:White; }
   /*将字体设置为非常小，从而达到隐藏目的*/
   .hiddenText2{ font-size:1px;}
```

```
    </style>
</head>
<body>
<div>
这一段是正文。正文结束。
<span class="hiddenText1">                             <!--下面是隐藏的关键字-->
ASP.NET C# jQuery Web Service SQL Server 编程 项目 案例 教程
</span> <br />
这是<span class="hiddenText2">asp.net</span>   <!--隐藏的关键字-->
另外<span class="hiddenText2">C#</span>        <!--隐藏的关键字-->
一段<span class="hiddenText2">案例</span>      <!--隐藏的关键字-->
正文<span class="hiddenText2">教程</span>      <!--隐藏的关键字-->
。正文结束。<br />
</div>
</body>
```

5. 重定向

使用刷新标记、CGI 程序、Javascript 或其他技术，当用户进入该页时，迅速自动跳转到另一个网页。重定向使搜索引擎与用户访问到不同的网页。下面是一个重写向的例子，这个页面中包含针对搜索引擎进行了优化的页面，这个页面仅被搜索引擎搜索到，不能被用户看到，访问此页面的用户被重定向到另外的页面。

```
<body>
<script type="text/javascript" language="javascript">
    window.location = "想让用户看到的页面的地址";
</script>
这是页面正文，是给搜索引擎看的，用户不会看到。
</body>
```

6. 复制站点或内容

通过复制整个网站或部分网页内容并分配以不同域名和服务器，以此欺骗搜索引擎对同一站点或同一页面进行多次索引的行为。一个最典型例子是镜像站点。

7. 门页

门页也称桥页，针对某一关键词专门制作一个优化的页面，链接指向或重定向到目标页面。有时候为动态页面建立静态入口，或为不同的关键词建立不同内页也会用到类似方法，但与门页不同的是，前者是网站实际内容所需而建立的，是访问者所需要的，而门页本身无实际内容，只针对搜索引擎作了一堆充斥了关键词的链接而已。

6.2　URL 重写优化

URL 是一个页面的“名片”，用户和搜索引擎都是通过 URL 访问页面。一个简洁、易

懂、易记的 URL 能够使页面更加友好，也能够提高搜索引擎对页面的评价。本节将介绍 URL 设计的原则，并结合 ASP.NET 编程技术说明如何通过重写 URL 实现优化。

6.2.1　静态 URL 和动态 URL

根据 URL 中是否包含检索字符串（即 URL 中问号及其后面的内容），可以将 URL 分成静态和动态两种类型。静态 URL 是指不包含检索字符串的 URL，静态 URL 通常具有更好的可读性，也更加容易记忆，下面是两个静态 URL 的例子。

```
http://www.mysite.com/home
http://www.mycompany.com/help/aboutme.html
```

动态 URL 是指包含检索字符串的 URL。这种 URL 在 ASP.NET 开发的 Web 项目中很常见，检索字符串通常用来给页面传递参数。以下是两个动态 URL 的例子。

```
http://www.mycompany.com/productdetail.aspx?pid=book1
http://www.mycompany.com/admin/userright.aspx?userid=admin&roleid=01
```

在 ASP.NET 开发的项目中，很多时候都需要使用同一个页面显示不同的内容，例如，在网上书店项目中，使用同一个页面 BookDetail.aspx 显示不同的图书信息。为了给 BookDetail.aspx 页面指定要显示的图书，通常需要在 URL 中添加一个检索字符串以指定图书编号，这样就形成了动态 URL。

对于搜索引擎来说，静态 URL 比动态 URL 拥有更高的价值。在使用 ASP.NET 开发 Web 网站时，考虑到搜索引擎优化的因素，应该尽量避免使用动态 URL。那么，如何处理检索字符串和静态 URL 之间的矛盾呢？URL 重写技术很好地解决了这个问题。

6.2.2　URL 重写概述

URL 重写是截取传入的 Web 请求并自动将请求重定向到其他资源的过程。执行 URL 重写时，通常会检查被请求的 URL，并基于 URL 的值将请求重定向到其他 URL。例如，在进行网站重组而将/people/目录下的所有网页移动到/info/employees/目录中时，需要使用 URL 重写来检查 Web 请求是否指向了/people 目录中的文件。如果请求指向/people/目录中的文件，则自动将请求重定向到/info/employees/目录中的同一文件。

利用 URL 重写技术，将一个静态 URL 重定向到一个动态 URL，可以避免让用户和搜索引擎看到动态 URL。例如，当用户请求 BookDetail/12 页面时，把这个静态 URL 重定向到 BookDetail.aspx?id=12 的页面，后者即为 ASP.NET 项目中的真实页面，可以通过检索字符串获得页面参数，并根据参数动态生成页面内容。而这一切对用户来说都是透明的，用户看到的是请求页面 BookDetail/12，就返回了编号为 12 的图书信息，请求另外一个页面 BookDetail/20，就返回编号为 20 的图书信息。

6.2.3　使用 HTTP 模块重写 URL

在 ASP.NET 程序中，处理每一个 HTTP 请求时都会执行已经注册的 HTTP 模块，所以在 HTTP 模块中可以编写适用于所有请求的公用代码。对于重写 URL 来说，在得到一个请求以后，要分析这个被请求的 URL，如果需要重写，则按照重写规则跳转到重写后的页面。这个任务适合在 HTTP 模块中实现。下面通过一个例子说明具体如何实现。

【例 6-1】 简单的 URL 重写。

在第 4 章所开发的网上书店项目中，有许多动态 URL，一个典型的例子是图书详情页面。当用户在图书列表中单击一本图书时，就会转到图书详情页面，并将所选择的图书编号作为检索字符串传递给图书详情页面，形成类似于 BookDetail.aspx?id=10 的动态 URL。为了将动态 URL 优化成静态 URL，希望使用 BookDetail/10 的 URL 访问刚才的页面，这就需要 URL 重写。当网上书店程序收到一个静态 URL 请求时，将其重写为动态 URL。

（1）打开第 4 章所创建的网上书店项目 BookShop。

（2）在项目中添加一个 HTTP 模块，命名为 MyUrlRewriter。在 HttpApplication 的 BeginRequest 事件中，检查所请求的 URL 并在必要时进行重写。代码如下：

```
public class MyUrlRewriter : IHttpModule
{
    public void Dispose()
    {   }
    public void Init(HttpApplication context)
    {
        context.BeginRequest += new EventHandler(context_BeginRequest);
    }
    void context_BeginRequest(object sender, EventArgs e)
    {
        HttpContext context = HttpContext.Current;
        string path = context.Request.Path.ToLower();          //得到请求路径
        if (path.StartsWith("/bookdetail/"))
        {
            //静态 URL 形式为/bookdetail/6，其中 6 为图书编号，重定向动态 URL
            string id = path.Substring(path.LastIndexOf('/') + 1);
            context.RewritePath("~/BookDetail.aspx?id=" + id, false);
        }
    }
}
```

（3）打开图书列表控件 BookListControl.ascx，其中的链接地址原为动态 URL，如下代码：

```
<a href="BookDetail.aspx?id=<%#Eval("BookID")%>">
<img src="BookImages/<%#Eval("SmallImage") %>" alt="<%#Eval("BookTitle")
%>" style ="width: 100px;" />
</a>
```

将上述代码中的动态 URL 改为静态 URL，代码如下：

```
<a href="BookDetail/<%#Eval("BookID")%>">
<img src="BookImages/<%#Eval("SmallImage") %>" alt="<%#Eval("BookTitle")
%>" style ="width: 100px;" />
</a>
```

（4）运行网上书店项目，从图书列表中选择一个图书查看详情，可以看到图书详情页面的 URL 已经变成了静态 URL。运行界面如图 6.1 所示。

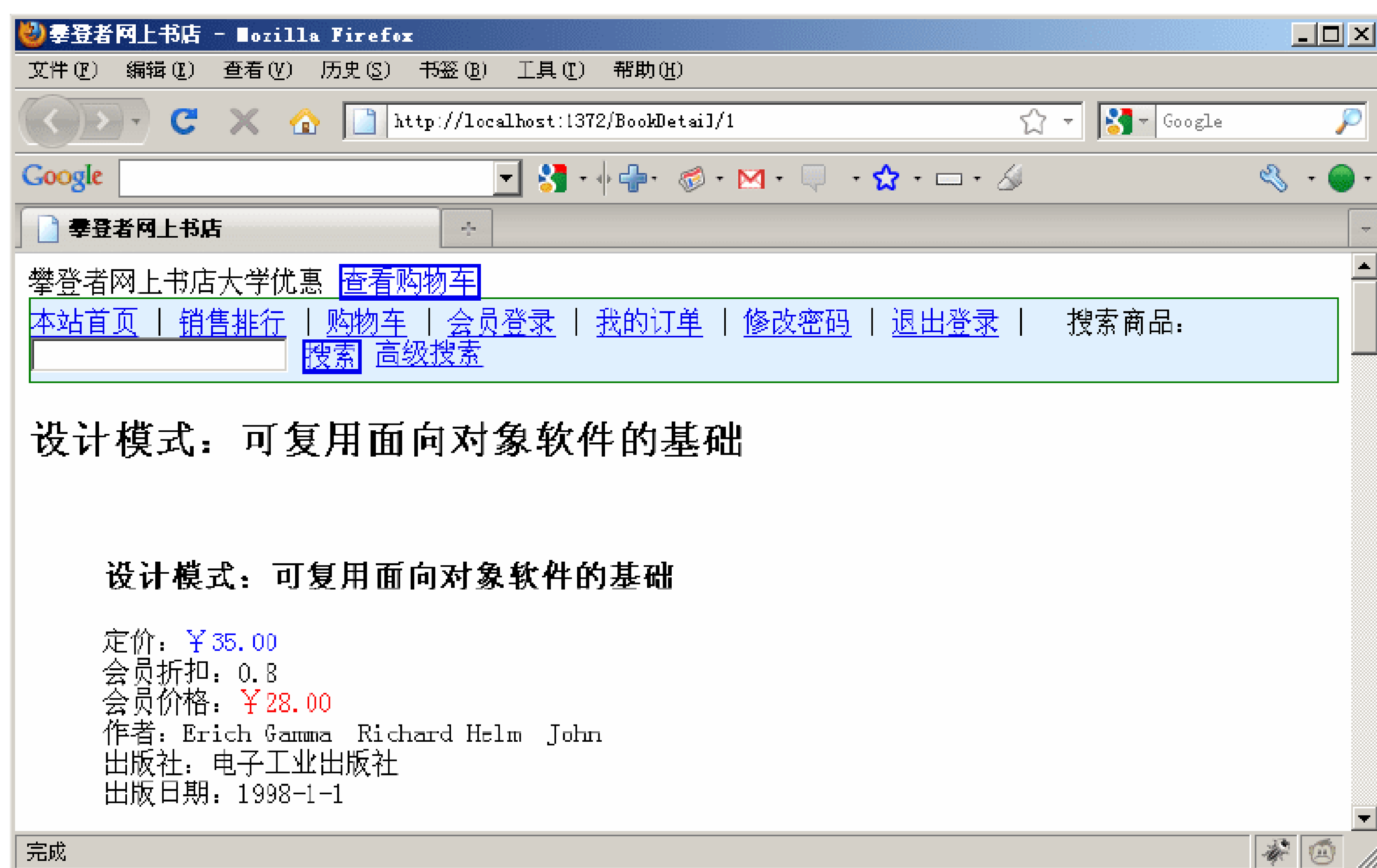


图 6.1　图书详情 URL 重写示例 1

（5）从图 6.1 中可以看到，页面上所有图片（包括顶部横幅、图书封面等）都不能显示了。这是由于在页面代码中使用相对路径引用图片，当 URL 重写后路径就发生了变化。例如，图书封面的相对路径为 BookImages/1b.jpg，在未进行 URL 重写以前，这个图片的绝对路径为 /BookImages/1b.jpg，而进行了 URL 重写后，图片绝对路径变为 /BookDetail/BookImages/1b.jpg。为了解决这个问题，所有图片应该使用绝对路径而非相对路径。在母版中修改 LOGO 和横幅图片路径，如下代码所示（注意图片路径是以/开头，表示绝对路径，如果没有这个/，则为相对路径）。

```
<img src="/images/logo.jpeg" alt="攀登者网上书店" style="height:80px;" />
<img alt="大学优惠" src="/images/banner.jpg" style="height:80px;" />
```

同样的道理，在 BookDetail.aspx 页面中需要修改图书封面为绝对路径。

```
cover.ImageUrl = "/BookImages/" + book.bigImage;
```

（6）如果页面上的 JavaScript 和 CSS 使用相对路径引用，则也存在与图片相同的问题，将所有 JavaScript 和 CSS 文件引用都修改为绝对路径。

（7）再次运行项目，并查看某图书详情，可以看到所有图片都能正常显示，所有 CSS 样式也正常应用，如图 6.2 所示。



图 6.2　图书详情 URL 重写示例 2

6.2.4　处理回发

URL 重写后，由于改变了页面的路径，会带来一系列的问题，例如找不到图片、外部 CSS 文件、外部 JavaScript 文件等，解决这个问题的方法是使用绝对路径引用这些文件，如例 6-1 中所采取的方案一样。除此以外，由于 URL 重写造成页面路径改变还会带来一个更加严重的问题，就是页面上的所有服务器端事件（如按钮的服务器端 Click 事件）都无法正常工作。在例 6-1 中，在图书详情页面上单击“加入购物车”按钮，则会出现如图 6.3 所示的错误界面。

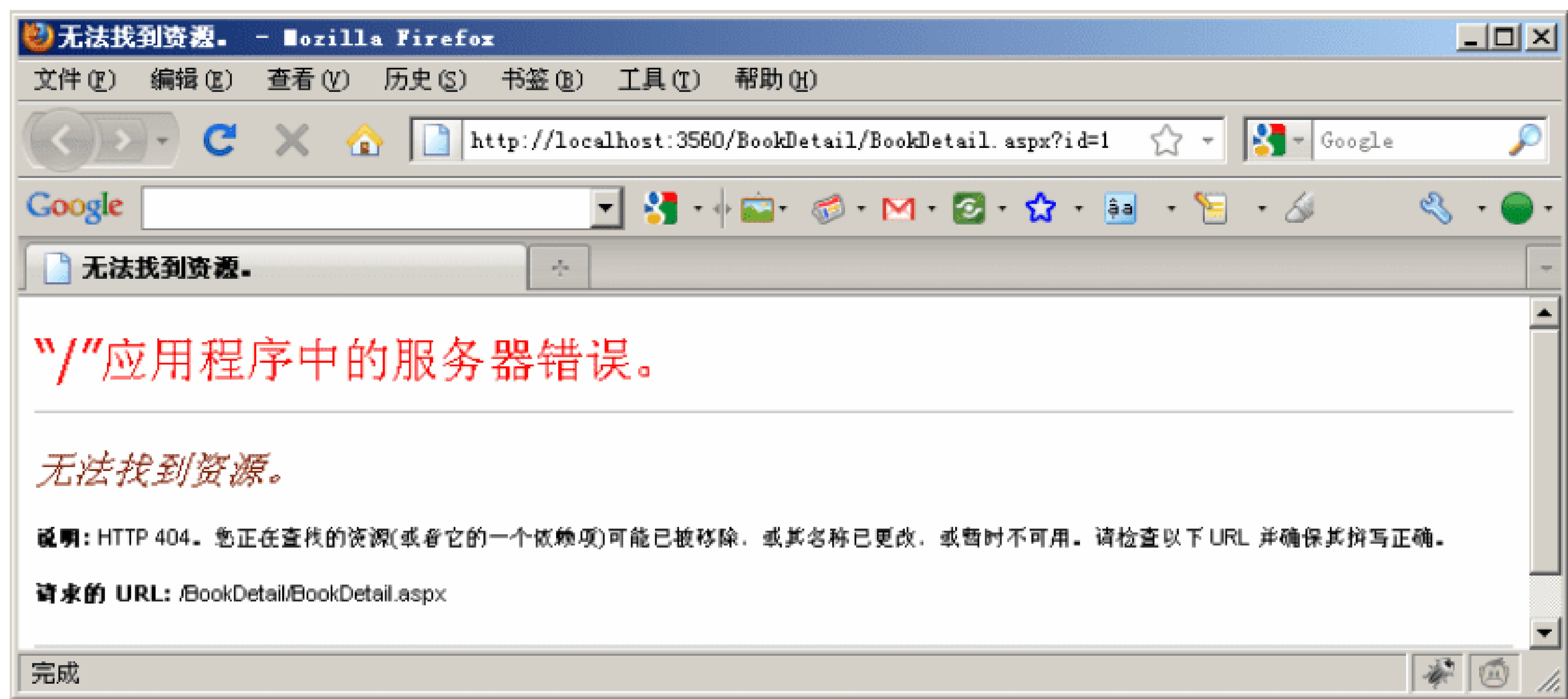


图 6.3　单击按钮后产生错误

在图 6.3 中，提示无法找到资源/BookDetail/BookDetail.aspx。从浏览器的地址栏中可以看到，在查看图书详情时，页面 URL 为/BookDetail/1（其中 1 为正在查看的图书编号），当单击“加入购物车”按钮后，页面地址变为/BookDetail/BookDetail.aspx?id=1。这种情况带来两个问题：首先是这个页面地址不存在，造成无法找到资源；其次是这个地址又变成

了 URL 优化以前的动态 URL，URL 优化失去了作用。

之所以会再现上述情况，是由于 URL 重写后，浏览器端所显示的路径与服务器端真实路径不同。具体来说，ASP.NET 的 Web 窗体会在所生成的 HTML 代码中自动添加一个 action 属性，这个属性指明了页面所要提交的地址。ASP.NET Web 窗体默认为自提交，即提交到页面本身。在例 6-1 中，由于 URL 重写，浏览器端比服务器端多了一级 BookDetail 路径。图书详情页面在提交时，仍然提交到自身，即 BookDetail.aspx，这个页面与浏览器端路径相结合，就得到了一个错误的路径/BookDetail/BookDetail.aspx，于是出现了图 6.3 所示的错误。

URL 重写后，不仅单击按钮会造成如图 6.3 所示的错误，其他服务器端事件也会出现同样错误。这是因为 ASP.NET 服务器端事件都是提交到服务器端页面本身，然后再用页台代码进行处理。由于 URL 重写后页面提交的地址不对，找不到要提交的页面，就会出现如图 6.3 所示的错误。

由于 URL 的重写造成页面无法提交，从而导致服务器端任何事件都不能工作，这是一个很严重的问题。解决这个问题的方法是使页面提交时仍然提交到经过重写优化以后的 URL。由于页面提交的地址是在页面生成的 HTML 文档中 form 元素的 action 属性中指定，要想修改提交地址，必须想办法修改页面所生成的 HTML 代码中的 action 属性。下面通过一个例子来说明具体实现步骤。

【例 6-2】 正确处理回发。

本例演示了如何解决 URL 重写带来的页面回发错误，基本思想是重写 form 元素的 action 属性，将其修改为经过优化以后的 URL（即浏览器端看到的 URL）。这个解决方案来源于 Scott Guthrie 的博客，其中用到了 ASP.NET 控件适配器，读者可以参考原文以获得更多信息：http://weblogs.asp.net/scottgu/archive/2007/02/26/tip-trick-url-rewriting-with-asp-net.aspx。

（1）打开网上书店项目。

（2）在项目中添加一个控件适配器类 UrlRewriterControlAdapter 和一个 UrlRewriter 类，这两个类协作完成 HtmlForm 类的自定义呈现，修改了最终生成的 HTML 代码中的 action 属性。代码如下：

```
public class UrlRewriterControlAdapter:ControlAdapter
{
    protected override void Render(HtmlTextWriter writer)
    {
        //使用自定义的 UrlRewriter 类呈现控件（此处为 HtmlForm）
        base.Render(new UrlRewriter(writer));
    }
}
public class UrlRewriter : HtmlTextWriter
{
    //构造函数
    public UrlRewriter(HtmlTextWriter writer)
        : base(writer)
    {
        this.InnerWriter = writer.InnerWriter;
    }

    /// <summary>
```

```
    /// 重写基类的 WriteAttribute()方法
    /// </summary>
    /// <param name="name">要写的 Attribute 的名称</param>
    /// <param name="value">要写入的 Attribute 的值</param>
    /// <param name="fEncode">是否对属性名称和值进行编码</param>
    public override void WriteAttribute(string name, string value,bool
    fEncode)
    {
        //检查当前要写入的属性是否为 action 属性
        if (name.ToLower() == "action")
        {
            var context = HttpContext.Current;
            if (context.Items["ActionRewrite"] == null)
            {
                value = context.Request.RawUrl;
                                        //将其重写为浏览器请求的 Url
                context.Items["ActionRewrite"] = true;
                                        //设置标记，已经重写 action 属性
            }
        }
        base.WriteAttribute(name, value,fEncode);
                                        //调用基类相应方法
    }
}
```

（3）在解决方案资源管理器中，在项目名称上右击，从弹出的快捷菜单中选择“添加”|“添加 ASP.NET 文件夹”|“App_Browsers”命令，添加一个 App_Browsers 文件夹。

（4）在 App_Browsers 文件夹中，添加一个浏览器文件 my.browser，并在其中注册刚才所创建的控件适配器类 UrlRewriterControlAdapter，代码如下：

```
<browsers>
   <browser refID="Default">
     <controlAdapters>
       <adapter controlType="System.Web.UI.HtmlControls.HtmlForm"
               adapterType="BookShop.UrlRewriterControlAdapter" />
     </controlAdapters>
   </browser>
</browsers>
```

（5）运行网上书店，转到图书详情页面，单击“加入购物车”按钮，可以看到，按钮能够正常工作，再转到查看购物车页面，可以看到商品已经被放入购物车中，从而说明这种方案成功解决了 URL 重写后的页面回发问题。

6.3　正则表达式与 URL 重写

在 6.2 的例子中，只实现了一个页面即 BookDetail.aspx 的 URL 重写，实现的代码简单直观，判断所请求的 URL 是否具有类似“/bookdetail/1”的形式，如果是，则将 URL 重写为动态 URL 的形式 BookDetail.aspx?id=1，代码如下：

```
HttpContext context = HttpContext.Current;
string path = context.Request.Path.ToLower();
                                        //得到请求路径
```

```
if (path.StartsWith("/bookdetail/"))
{
    //静态 URL 形式为/bookdetail/1，其中 1 为图书编号，重定向动态 URL
    string id = path.Substring(path.LastIndexOf('/') + 1);
    context.RewritePath("~/BookDetail.aspx?id=" + id, false);
}
```

在较大型的网站和项目中有很多页面需要重写 URL，如果都采用上述方法，则每个页面都需要写一个 if 或者 switch 分支，代码很复杂。而且，每当添加一个新的需要重写的 URL，都需要修改代码，增加一个 if 或者 switch 分支。这种 URL 重写方式不灵活，经常需要修改，不是一种好的设计。使用正则表达式可以实现更加通用的解决方案。

6.3.1 正则表达式语法

正则表达式是指一个用来描述或者匹配，一系列符合某个句法规则的字符串的单个字符串。一个正则表达式通常被称为一个模式（pattern），被广泛用于各种字符串处理中，如查找、替换、格式验证等。正则表达式定义了一个字符串模式，例如，规定了字符串中应该包含哪些字符，不能包含哪些字符，以什么字符开头和结尾，某些字符出现多少次等。正则表达式本身也是一个字符串，用一种特定语法描述模式，其是区分大小写的。组成正则表达式的元素大致有以下几种。

1. 普通字符

普通字符表示字符本身。例如，模式“abc”表示包含 abc 的任意字符串。

2. 转义字符

正则表达式中的转义字符与 C#字符串中的转义字符含义类似。转义字符可以看成是普通字符的一种特殊形式。例如，模式“a\tb”匹配任意一个字符 a 后面紧跟一个水平制表符再紧跟一个字符 b。

3. 元字符

元字符可以匹配一类字符，例如匹配数字、字符、空白等。常用的元字符及其含义如下。

- [...]：匹配出现于[]中的任意单个字符。
- [^...]：匹配不在[]出现的任意单个字符。
- .：匹配除换行符以外的任意单个字符。
- \w：匹配任意单个字母、数字、下划线、汉字。
- \W：匹配任意单个不属于\w 模式的字符。
- \s：匹配任意单个空白字符。
- \S：匹配任意单个非空白字符。
- \d：匹配单个数字字符，相当于[0-9]。
- \D：匹配任意单个非数字字符，相当于[^0-9]。

提示：如果要在字符串中匹配小数点“.”，则需要在模式中写成“\.”的形式，即在“.”的前面添加“\”将其转义成普通字符。如果直接使用“.”进行匹配，那么可以匹配除换行符以外的任意单个字符。

4. 开头和结尾

在定义正则表达式的模式时，有两个特殊字符^和$，其中^表示匹配字符串开头，$表示匹配字符串结尾。例如，模式^[A-Z]匹配以大写英文字符开头的字符串，模式\.$匹配以英文句号（小数点）结尾的字符串。

提示：在正则表达式中，^字符有两种完全不同的含义。当^出现于方括号[]内时，表示“非”，匹配不出现在方括号[]中的任意字符。当^出现在方括号[]外时，表示字符串开头。

5. 重复

在正则表达式中还可以指定元素出现的次数，有以下几种方式。

- {m,n}：匹配 m 到 n 次重复（闭区间）。例如 a{1,3}匹配 1 到 3 个字符 a。
- {m,}：匹配大于等于 m 次重复。例如 a{2,}匹配 2 个以下字符 a。
- {m}：匹配正好 m 次重复。
- ?：匹配 0 到 1 次重复，相当于{0,1}。这种重复也称为可选，即可以出现也可不出现。
- +：匹配 1 次以上重复，相当于{1,}。
- *：匹配 0 次以上重复，相当于{0,}。

6. 选择

正则表达式使用符号|表示“或”的关系。例如 a|A 表示小写字母或者大写字母 a。

7. 分组和引用

正则表达式使用圆括号（）将多个字符分成一组成为逻辑上的一个单位。例如，模式(\+0)?123 表示括号里面的+0 是可选的（其中+前面的\为转义符），而\+0?123 则指 0 是可选的。字符串+0123 和 123 都能够匹配这两个模式，字符串+123 能够匹配第 2 个模式但不能匹配第 1 个模式。

在正则表达式中还经常需要引用前面的内容，例如，模式\d{3}可以表示一个三位数，那么如果要表示这个三位数重复 2 次应该如何写呢？显然不能用\d{3}\d{3}，这个模式其实表示 6 个数字，或者说任意一个 6 位数。如果要表示一个 3 位数重复 2 次，那么就需要在第 1 个 3 位数后面再出现与前一个 3 位数完全相同的内容，也就是说，在模式中需要引用前面出现的内容。

正则表达式提供的分组和引用机制可以给模式中出现的一部分内容分组，并且在后面的模式中引用这个分组。如前所述，正则表达式使用圆括号（）进行分组，还可以为分组指定一个名称，语法为在紧跟在左侧圆括号后面写上一个问号和一对包括在尖括号中的名

字。这种分组称为命名分组，在模式中可以通过分组名称引用某个分组。如刚才所说的一个 3 位数重复出现 2 次可以用以下模式来描述：

```
(?<group1>\d{3})\k<group1>
```

正则表达式中的分组也可以是匿名分组。如果在分组的左侧圆括号后面没有写问号、尖括号和组名，则这个分组为匿名分组。引用匿名分组的方法是使用分组序号，整个模式中第 1 个分组的序号为 1，第 2 个分组的序号为 2，依次类推。可以通过\1 的方式引用第 1 个分组。例如，刚才所说的 3 位数重复出现 2 次的例子也可以用以下模式来描述：

```
 (\d{3})\1
```

6.3.2　正则表达式验证

正则表达式有很多用途，其中的一个常用功能就是验证，即验证某些文本是否符合某种模式。在.NET Framework 中，使用 System.Text.RegularExpressions 命名空间中的 Regex 类表示一个正则表达式，通过在 Regex 类的构造函数中传递一个字符串参数指定正则表达式模式，代码如下：

```
Regex r = new Regex(@"\d{6}");
```

Regex 类的 Match 方法执行匹配并返回一个 Match 类型的结果，Match 类的 Success 属性表示匹配是否成功，Match 类的 Value 属性表示匹配成功的文本，Match 类的 NextMatch 方法执行下一次匹配并返回匹配结果。下面是一个简单的例子，查找字符串中出现的数字并输出。

```
Regex r = new Regex(@"\d+");                    //创建正则表达式
Match m = r.Match("123aa45bb7cc89");            //执行第一次匹配
while (m.Success)                               //当匹配成功时循环
{
    Console.WriteLine(m.Value);                 //输出当前匹配的文本
    m=m.NextMatch();                            //执行下一次匹配
}
/*    输出结果为:
123
45
7
89      */
```

如果读者仔细观察上面的例子，就会发现一个问题：正则表达式的模式为\d+，表示匹配一到多个连续数字，如果目标字符串中包含多个连续数字，那么这个模式是只匹配第 1 个数字还是匹配全部数字呢？根据\d+模式的含义，匹配 1 个或者 2 个或者更多直到全部数字都是正确的，但是在实际执行时必须返回一个匹配结果。应该返回最小的匹配结果还是最大的匹配结果，这就是所谓的“贪婪”问题。

在正则表达式的重复模式中，默认使用贪婪方式进行匹配，就是说，要匹配尽可能多的次数。例如在前面的例子中，\d+总是匹配最多的连续数字。但是贪婪匹配也不总是最贪

婪的，也要考虑到后面其他内容的匹配，考虑下面的例子。

```
Regex r = new Regex(@"\w+end$");
Match m = r.Match("begin to end");
if (m.Success)
{
    Console.WriteLine("匹配成功，贪婪匹配并不是无限制的贪婪");
}
else
    Console.WriteLine("贪婪匹配太贪婪，把@也匹配了，导致整个正则表达式匹配失败");
```

在上面这段代码中，第一行定义了一个模式用来以 end 结尾文字，模式中的\w+可以匹配 1 到多个字母、数字、下划线、汉字，当然也可以匹配 end。如果贪婪匹配把 end 也匹配到\w+中，那么代码中定义的\w+end$就是永远不可满足的。而上面的代码运行的结果却显示匹配成功，说明贪婪匹配在最大限度的匹配更多字符时，也要受其后模式的限制，或者说，贪婪匹配是在满足整个模式匹配成功的基础上，去匹配尽可能多的重复次数。

与贪婪相反的另外一种模式匹配称为非贪婪匹配，非贪婪匹配总是尽可能匹配最少的重复次数。通过在重复次数后面添加一个问号指定该重复使用非贪婪匹配，如下例所示。

```
Regex r = new Regex(@"\d+?");                    //使用非贪婪匹配
Match m = r.Match("123aa45bb7cc89");             //执行第一次匹配
while (m.Success)                                //当匹配成功时循环
{
    Console.WriteLine(m.Value);                  //输出当前匹配的文本
    m=m.NextMatch();                             //执行下一次匹配
}
```

上面的代码运行时，会输出九行分别显示 1 到 9，说明每次\d+?模式只匹配了一个数字，即满足模式的最小重复次数。与贪婪匹配类似，非贪婪匹配也要受整体模式匹配的限制，为了使整个模式匹配成功，非贪婪模式在必要时也会匹配更多重复次数。

使用 Regex 进行字符串模式匹配时，默认情况是区分大小写，如下代码会匹配失败。

```
Regex r = new Regex(@"end$");                    //此模式要求以 end 结尾
Match m = r.Match("TheEnd");                     //待检验文本以 End 结尾
if (m.Success)
{
    Console.WriteLine("匹配成功");
}
else
    Console.WriteLine("匹配失败");               //由于大小写不一致匹配失败
```

如果要使 Regex 忽略字符大小写，则需要指定一个选项。Regex 类有一个 Options 属性，这是一个 RegexOptions 枚举，描述了正则表达式匹配时的一些选项，如是否区分大小写、是否从右向左查找匹配、是否执行多行模式等。如果要忽略字母大小写，可以在 Regex 构造函数中传递一个 RegexOptions.IgnoreCase 值，代码如下：

```
Regex r1 = new Regex("end$", RegexOptions.IgnoreCase);
```

下面通过一个综合性的例子演示如何通过 Regex 验证常用的字符串模式。

【例 6-3】 正则表达式验证。

本例演示使用 Regex 验证常用的字符串模式。

（1）创建一个控制台应用程序 RegexSample。

（2）在项目中添加一个类 RegexTest 用以测试正则表达式，类的代码如下：

```
//测试 Regex 的类
class RegexTest
{
    //测试数据类
    class RegexTestData
    {
        public string pattern;                          //正则表达式模式
        public string description;                      //模式的说明
    }
    public void test()
    {
        List<RegexTestData> list = new List<RegexTestData>();
        //中国邮箱由 6 位数字组成
        RegexTestData test = new RegexTestData()
        {
            pattern = "^[0-9]{6}$",
            description = "中国邮政编码"
        };
        list.Add(test);
        //中国电话号码: 3～4 位区号 + 横线- + 7～8 位市话号码
        test = new RegexTestData()
        {
            pattern = @"^(\d{3}-|\d{4}-)?\d{8}|\d{7}$",
            description = "中国电话号码"
        };
        list.Add(test);
        //URL 地址：多个字母开头，后跟://, 然后是多个非空白字符
        test = new RegexTestData()
        {
            pattern=@"[a-zA-z]+://[^\s] *",
            description="URL 地址"
        };
        list.Add(test);
        /* 电子邮箱地址模式分析:
            * 1 到多个字母数字下划线开头， 中间可以有-+.符号， 必须有@符号
            * @号后面的内容被.分成两部分，每一部分的模式与@号前面的模式基本相同*/
            test = new RegexTestData()
        {
            pattern=@"\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*",
            description="电子邮件地址"
        };
        list.Add(test);
        for (int i = 0; i < list.Count; i++)
```

```
            {
                Console.Write("请输入一个{0}:",list[i].description);
                string s = Console.ReadLine();
                testMatch(s, list[i].pattern);
            }
        }
        /// <summary>
        /// 测试正则表达式是否匹配
        /// </summary>
        /// <param name="text">要测试的文本</param>
        /// <param name="pattern">用来测试的模式</param>
        private void testMatch (string text,string pattern)
        {
            Regex r = new Regex(pattern);
            Match m = r.Match(text);
            if (m.Success)
                Console.WriteLine("匹配成功，结果为:"+m.Value);
                                                    //匹配成功则输出匹配结果
            else
                Console.WriteLine("匹配失败");
        }
    }
```

（3）在 Main()方法中调用以上测试方法。

```
static void Main(string[] args)
{
    new RegexTest().test();
    Console.ReadKey();
}
```

（4）运行此程序，运行结果如下：

```
请输入一个中国邮政编码:100123
匹配成功，结果为:100123
请输入一个中国电话号码:021-12345678
匹配成功，结果为:021-12345678
请输入一个 URL 地址:http://test.com
匹配成功，结果为:http://test.com
请输入一个电子邮件地址:my-this@163.com
匹配成功，结果为:my-this@163.com
```

6.3.3　正则表达式查找和替换

正则表达式可以实现复杂的字符串查找和替换。在.NET Framework 中，使用正则表达式进行字符串查找时，除了 Regex 类以外，还需要用到 Capture、Group 和 Match 类。Capture 类表示一个捕获结果有 3 个重要属性：Index 表示捕获的字符串在整个字符串中的位置，Length 表示所捕获字符串的长度，Value 表示所捕获字符串的值。Group 类表示模式匹配中

的一个组，Group 类从 Capture 类继承。Match 类表示一次匹配，Match 类从 Group 类继承。Group 类有一个 CaptureCollection 类型的 Captures 属性，表示这一组所有捕获的集合。Match 类有一个 GroupCollection 类型的 Groups 属性，表示这个匹配的所有组的集合。这几个类之间的关系如图 6.4 所示。

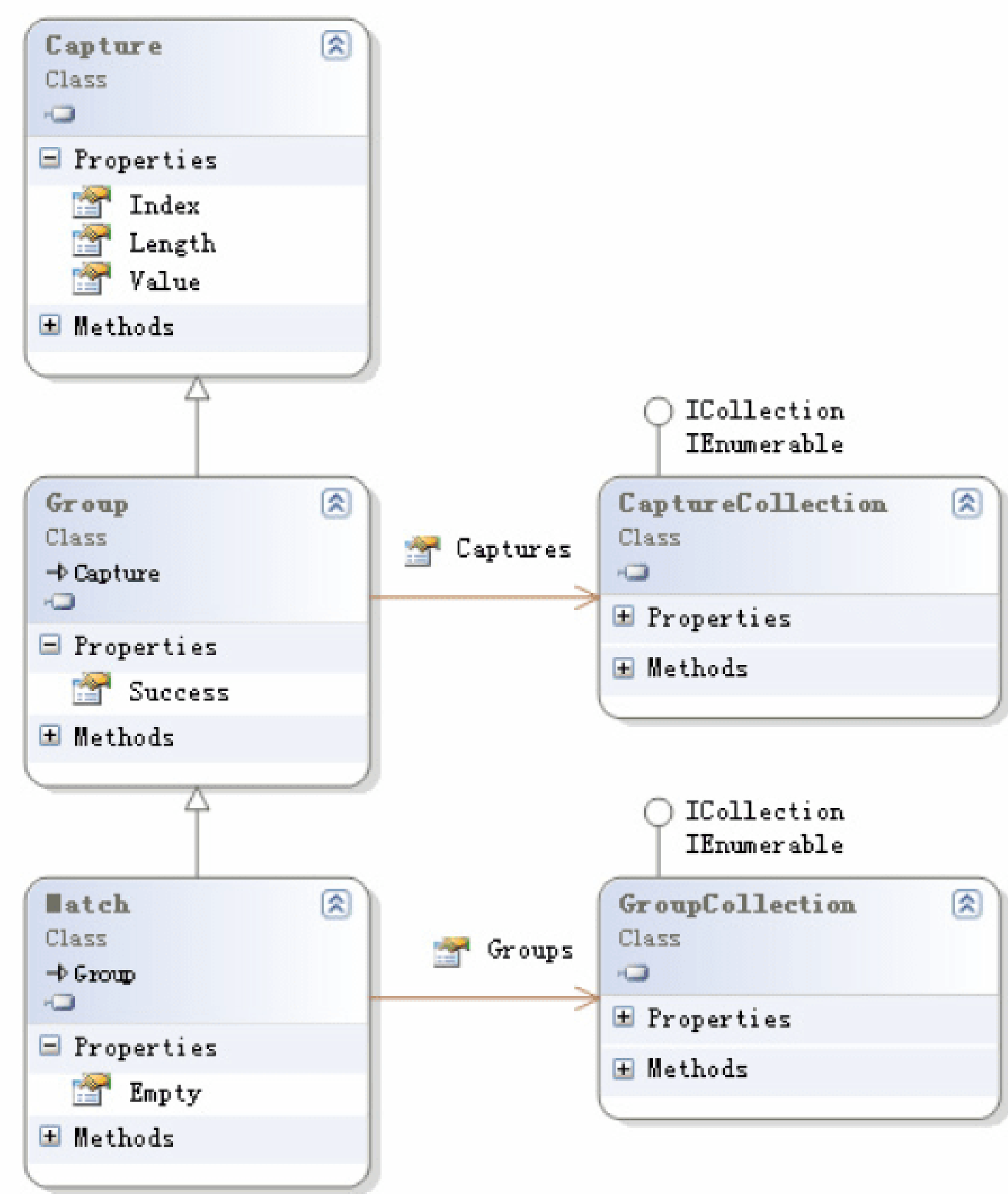


图 6.4　正则表达式类关系图

Capture、Group、Match 这几个类之间的关系较为复杂，既有继承关系又有集合关系。通过正确理解并合理使用这些类，可以完成复杂的字符串操作。下面通过两个例子来演示这几个类的关系及应用。

【例 6-4】 捕获、组和匹配。

本例演示捕获 Capture、组 Group 和匹配 Match 类的应用及其关系。

（1）创建一个控制台应用程序，在 Main()方法中编写以下代码。

```
static void  Main()
{
   int count;
   Regex regex = new Regex("(123)+");                //匹配一到多个 123
   string text = "abc123123123abc123123abc";         //要查找的字符串
   Match match = regex.Match(text);                  //执行一次匹配
   while (match.Success)                             //若匹配成功则循环
   {
        Console.WriteLine("找到一个新的匹配");
        GroupCollection groups = match.Groups;
                                                     //获得此次匹配组的集合
        Console.WriteLine("该匹配共有{0}组",groups.Count);
        for (int i = 0; i < groups.Count; i++)
        {
             Group g = groups[i];
             Console.WriteLine("\t 新的一组");
             CaptureCollection captures = g.Captures;
```

```
                                                    //获得当前组的所有捕获
        count = captures.Count;
        Console.WriteLine("\t 该组中共有{0}个捕获", count);
        for (int j = 0; j < count; j++)
        {
            Capture c = captures[j];
            //显示一个捕获的详细信息
            Console.WriteLine("\t\t 捕获详情: 位置:{0}, 文本:{1}",c.Index,
            c.Value);
        }
    }
    match = match.NextMatch();                      //尝试下一个匹配
  }
}
```

（2）运行上述程序，输出结果如下：

```
找到一个新的匹配
该匹配共有 2 组
    新的一组
    该组中共有 1 个捕获
        捕获详情: 位置:3, 文本:123123123
    新的一组
    该组中共有 3 个捕获
        捕获详情: 位置:3, 文本:123
        捕获详情: 位置:6, 文本:123
        捕获详情: 位置:9, 文本:123
找到一个新的匹配
该匹配共有 2 组
    新的一组
    该组中共有 1 个捕获
        捕获详情: 位置:15, 文本:123123
    新的一组
    该组中共有 2 个捕获
        捕获详情: 位置:15, 文本:123
        捕获详情: 位置:18, 文本:123
```

【例 6-5】 HTML 文本提取。

本例演示如何从 HTML table 中去除<tr><td>等标记，读取其中的文字。

问题分析：根据对 HTML 中 table 的格式可知，table 中能够显示在浏览器中的内容就是<td>标记中的内容。因此，要提取 table 中的内容，只需取出<td>和</td>标记之间的文本即可。用正则表达式很容易实现这个功能。

（1）创建一个控制台应用程序，并编写如下代码。

```
static void getHtml()                               //版本 1
{
    //原始 HTML 字符串
    string table = "<table>"
        +"<tr><td>one</td><td>two</td></tr>"
        +"<tr><td>three</td><td>four</td></tr>"
        +"</table>";
    Regex regex = new Regex("<td>(.)+</td>");
                                                    //正则表达式
    Match match = regex.Match(table);
    while (match.Success)
```

```
    {
        Console.WriteLine(match.Value);
        match = match.NextMatch();
    }
}
```

（2）运行上述代码，但是并没有得到期望的输出，而是输出以下内容：

```
<td>one</td><td>two</td></tr><tr><td>three</td><td>four</td>
```

观察上述输出结果，可以看出匹配的整个字符串的第一个<td>和最后一个</td>之间的内容，这是由于贪婪匹配造成的。正则表达式默认使用贪婪匹配，总是匹配尽可能多的字符，而定义的模式为以<td>开头、以</td>结尾的字符串，按照贪婪匹配原则，就匹配了最开始的<td>和最末尾的</td>。

（3）为了解决贪婪匹配带来的问题，正确取出<td>和</td>之间的内容，需要使用非贪婪匹配，对前面的代码做以下修改（只给出修改部分）。

```
Regex regex = new Regex("<td>(?<g1>.+?)</td>");    //使用非贪婪匹配，命名分组
Match match = regex.Match(table);
while (match.Success)
{
    Console.WriteLine(match.Groups["g1"].Value);   //输出命名分组中的内容
    match = match.NextMatch();
}
```

（4）再次运行程序，这次输出了正确结果如下：

```
one
two
three
four
```

在处理字符串时，除查找功能外，也经常用到替换功能。例如，在 URL 搜索引擎优化中，需要把一个动态 URL 重写为静态 URL，从动态 URL 到静态 URL 的转换就是一个字符串替换的过程。Regex 类提供了一个 Replace()方法可以实现字符串替换，方法签名如下。

```
public string Replace(string input,string replacement);
```

Regex 类 Replace()方法的功能在指定的输入字符串内，使用指定的替换字符串替换与某个正则表达式模式匹配的所有字符串。该方法第一个参数 input 为要替换的原始字符串，replacement 参数为替换字符串，方法返回替换以后得到的字符串。例如，下面的代码把字符串中的多个连续空格替换为一个空格。

```
Regex regex = new Regex(@"\s+");
string text = "this is   the   original     text";
string newText=regex.Replace(text, " ");
```

在字符串替换时，有时候需要用前面出现的字符串来替换后面的某些字符串，例如，英语中姓名有两种写法，名在前或者姓在前，当姓在前名在后时需要在姓和名之间添加一个逗号。例如，比尔盖茨的名字可以写作 Bill Gates 或者 Gates, Bill 都可以。如果要把一个人的英文名字从名在前的格式替换为姓在前的格式，则可以使用以下代码实现。

```
string input="Sun Ji-Lei";
//定义模式（其中包含两个命名分组）
Regex regex=new Regex(@"(?<first>\S+) (?<last>\S+)");
//执行替换，使用{$组名}方式引用原始字符串中的分组
string output = regex.Replace(input,@"${last},${first}");上述代码
```

6.3.4　正则表达式 URL 重写

使用正则表达式强大的匹配替换功能，可以实现更加灵活通用的 URL 重写。其基本原理也是使用 HTTP 模块捕获所有访问请求，将请求的 URL（通常是静态 URL）按照一定的模式进行替换，得到一个新的 URL（通常是动态 URL），然后访问这个重写后的 URL。

根据以上理论，如果要自己编码实现利用正则表达式重写 URL，需要在例 6-1 的基础上，在 HTTP 模块中定义重写 URL 所使用的正则表达式，并使用这些正则表达式对静态 URL 进行替换即可。目前网络上也有几个开源的 URL 重写工具，如 UrlRewriter.NET（网址 http://urlrewriter.net/）、UrlRewritingNet（网址 http://www.urlrewriting.net/149/en/home.html）等，都支持正则表达式。这两个 URL 重写工具都使用配置文件定义重写的正则表达式。下面以 UrlRewriter.Net 为例说明如何实现 URL 重写。

【例 6-6】 UrlRewriter.Net 实现 URL 重写。

假设有一个面向全球各个国家的销售网站，使用多种语言向客户展示商品信息。网站根目录下有一个 Detail.aspx 页面，用来显示商品信息。访问此页面时需要指定两个参数：商品编号和界面语言。动态 URL 使用检索字符串传递参数，具有类似这种形式：Detail.aspx?language=en&id=10，其中 en 表示语言为英语，10 表示要查看的商品编号为 10。为了实现 URL 优化，希望用户使用静态 URL 访问这个页面，静态 URL 具有以下形式：/en/Detail10.aspx。本例演示如何使用 UrlRewriter.NET 实现这个功能。

（1）在 ASP.NET 项目中添加对 Intelligencia.UrlRewriter 程序集的引用。

（2）在 web.config 文件的 configSection 节中添加以下代码，这些代码的作用是告诉 UrlRewriter.NET 从 web.config 文件的 rewriter 结点中读取配置信息。

```
<configSections>
   <section name="rewriter"  requirePermission="false"
     type="Intelligencia.UrlRewriter.Configuration. RewriterConfiguratio-
     nSectionHandler, Intelligencia.UrlRewriter" />
 </configSections>
```

（3）在 web.config 文件中注册 UrlRewriter.NET 所使用的 HTTP 模块。

```
<httpModules>
<add  name="UrlRewriter"
 type="Intelligencia.UrlRewriter.RewriterHttpModule, Intelligencia.
 UrlRewriter"/>
</httpModules>
```

（4）在 web.config 文件中用添加用正则表达式描述的 URL 重写规则。

```
<rewriter>
<rewrite url="~/(.*)/Detail(.*).aspx" to="~/Detail.aspx? language=$1&amp;
id=$2" />
</rewriter>
```

（5）为了测试 URL 重写效果，在项目中添加一个页面 Detail.aspx。出于演示目的，本页并不根据参数显示不同的语言界面和产品信息，而只是在页面上显示从 QueryString 获得的语言和产品编号参数。

```
<form id="form1" runat="server">
<div>
你所使用的语言为：<asp:Label Text="" runat="server" ID="language"/> <br />
你所要查看的产品为：<asp:Label Text="" runat="server" ID="product" /><br />
</div>
</form>
```

（6）在 Detail.aspx.cs 文件中，在 Page_Load 事件中显示当前的语言和产品编号。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        product.Text = Request.QueryString["id"];
        language.Text = Request.QueryString["language"];
    }
}
```

（7）在网站中添加一个 HTML 页面，页面中放置几个超链接，以打开产品详情页面。

```
<body>
<a href="en/detail01.aspx">en,产品 01</a><br />
<a href="zh/detailp10.aspx">中文,产品 p10</a><br />
</body>
```

（8）运行网站，运行结果如图 6.5 所示。

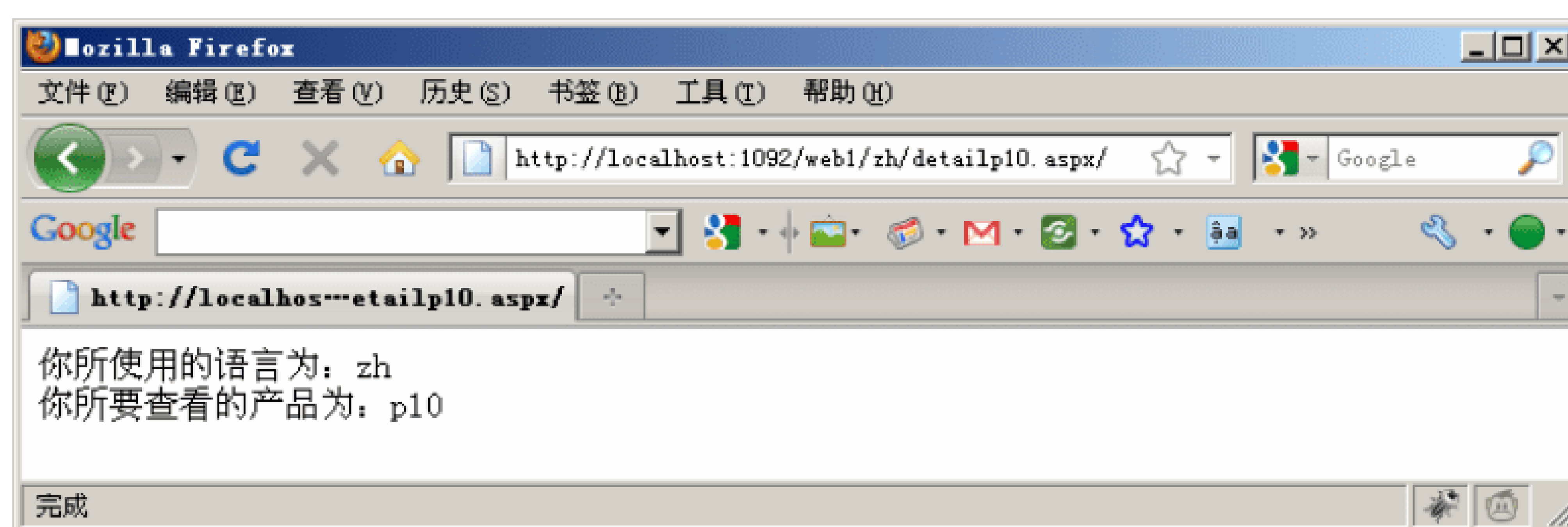


图 6.5　URL 重写效果

6.4　页面内容优化

对于搜索引擎优化而言，URL 优化仅是较简单的一步，更重要的还是要对页面内容进行优化。页面内容优化要比 URL 优化更加复杂，本节将介绍页面内容优化的主要原则和方法。

6.4.1　页面代码优化

页面代码优化是指通过合理编写页面代码（包括 HTML、JavaScript 等），达到突出页面主题和主要内容，减少页面文件大小的目的。页面代码的优化主要包括以下几个方面。

1. 合理使用关键字

搜索引擎检索页面信息时，根据页面内容来判断页面是否与所搜索的关键字匹配，关键字在页面上出现的次数越多，一般来说就意味着页面与被搜索关键字相关程度越高。当然，如本章 6.1 节所述，关键字的出现必须是合理的，如果堆砌关键字就成为搜索引擎作弊，会受到搜索引擎的惩罚。

关键字应该在页面代码的<head>部分和<body>部分都出现。在<head>部分，可以在<title>中为页面指定一个包含关键字的标题，在<meta>中指定页面的描述信息和关键词。在<body>部分，即页面显示的正文中，也要注意让关键字合理出现。例如，下面是一个介绍 ASP.NET 编程案例的页面，较好地使用关键字说明了页面的主题。

```
<head runat="server">
    <meta name="description" content="介绍了 ASP.NET 编程知识和一个真实项目案
    例" />
    <meta name="keywords" content="ASP.NET,编程,案例,教程" />
    <title>ASP.NET 项目案例</title>
</head>
<body>
    <h1>ASP.NET 项目教程之案例篇</h1>
     …
</body>
```

2. 消除冗余代码

当使用所见即所得的可视化开发工具（如 Visual Studio、Dreamweaver 等）编辑页面时，会产生一些冗余代码。页面中包含这些代码不但没有任何作用，反而增加页面大小，扰乱页面结构。例如当使用 Visual Studio 插入和编辑表格后，会产生许多空格字符和样式信息，如下代码所示。

```
<style type="text/css">
    .style1 {width: 82px;}
    .style2{width: 95px;}
    .style3{width: 83px;}
</style>
<table>
    <tr>
      <td class="style1">
            学号</td>
        <td class="style2" >
            姓名</td>
        <td class="style3">
            年龄</td>
    </tr>
    <tr>
        <td class="style1" >
```

```
             </td>
        <td class="style2">
             </td>
        <td class="style3">
             </td>
    </tr>
</table>
```

上述代码中，<td>标记中的 是多余的，从第 2 个<tr>开始，其中的 class="style1"也是多余的。把这些冗余代码删除以后，一方面减小了页面字节数，另一方面使页面更加清晰易读。

3. 突出重点内容

无论对于搜索引擎还是用户而言，页面中的某些内容显得比其他内容更为重要，例如，页面中的标题（在各级<h>标记中的内容）、页面上用粗体显示的字体等。当人们在浏览页面时，这些元素会受到更多重视，搜索引擎也模拟人的这一行为，认为这些元素更能说明页面内容。根据这一点，应该合理使用标题和加粗字体，突出页面主题，而不能整个页面都使用一样的文本格式。

4. JavaScript导航优化

搜索引擎是通过页面之间的链接对整个互联网进行搜索的。如果一个页面没有任何外部链接，那么这个页面不可能被搜索引擎蜘蛛访问到，更不可能被检索收录。目前，搜索引擎只识别<a href="somepage">标记为链接标记，根据 href 属性访问到所指向的页面。在设计网站时，大多数链接都是通过<a>实现的，但是也存在例外情况，最常见的就是有时候需要用 JavaScript 打开一个窗口，如下代码所示。

```
<a href="#" onclick="window.open('mypage.aspx','mywin');" >ASP.NET 项目案
例</a>
```

对于搜索引擎的蜘蛛程序来说，认为上述代码中<a>标记没有目标链接页面，从而无法访问到 mypage.aspx。可是如果直接把页面地址写到 href 属性中，又无法实现弹出窗口的效果。这时可以采用一个小技巧，就是一方面在 href 属性中包含要链接的真实地址，这样搜索引擎就能访问到目标页面，别一方面在 onclick 事件中用 window.open 打开目标窗口，然后返回 false，以避免打开<a>元素的目标地址，如下代码所示。

```
<a  href="#"  onclick="window.open('mypage.aspx','mywin');return  false;"
>ASP.NET 项目案例</a>
```

6.4.2　消除重复内容

如本章第 1 节所述，如果 2 个或多个页面包含基本相同的内容，会降低搜索引擎对页面的评价。由于各种原因，有些重复页面是不可避免的，最常见的情况就是打印机友好（print friendly）页面。

大多数的网页设计是为了在计算机屏幕上观看的。然而，有的时候用户需要将某些页面打印出来，也许就是为了保留一个长期的记录，或者将其用作方便的离线参考资料。但

随之而来的问题是，显示器是彩色的，而大多数打印机都是黑白的，在显示器上看起来引人注目和五彩缤纷的很多特性，都无法在打印机上表现出相同的效果。在被降级为灰度打印的时候，彩色的组合会失去原有的对比效果；图形会看起来失真，而且耗费太长的打印时间；在 Web 页面上起着重要作用的导航按钮在打印页面上也毫无用处。

为了克服这些问题，网页开发人员常常会为页面专门设计一个打印机友好的版本。打印机友好的版本通常都包括有和主要 Web 页面相同的内容，但是会省略掉大多数的图形、背景和导航元素。页面还会把彩色转换成某种形式，以便生成能够让人接受的灰度图像。

打印机友好页面很好的解决了显示和打印页面的矛盾，但是却在搜索引擎优化方面带来了另外一个问题，就是页面内容重复。如前所述，显示页面和打印机友好页面上的内容几乎是相同的，只是显示样式和图片不同。从搜索引擎优化的角度来说，希望搜索引擎只能看到其中的一个页面，通常是显示页面。为了达到这个目的，可以在页面中添加一个<meta>标记以拒绝搜索引擎对此页面的索引，如下代码所示。

```
<meta name="robots" content="noindex,nofollow" />
```

上述代码中的 robots 表示这个内容是针对搜索引擎蜘蛛程序的，noindex 表示不索引当前页面内容，nofollow 表示不跟踪当前页面中的链接。

6.5　小　　结

作为一个 ASP.NET 开发人员，需要掌握基本的搜索引擎优化知识，使开发的网站具有较好的搜索引擎评价。本章介绍了搜索引擎优化的基础知识，先介绍了搜索引擎优化的原理和误区，然后重点介绍了两种搜索引擎优化手段：URL 优化和页面内容优化。

第 2 篇　开发工具与第三方框架

- 第 7 章　Visual Studio 2010 新特性
- 第 8 章　LINQ 与实体框架 Entity Framework
- 第 9 章　ASP.NET AJAX 框架
- 第 10 章　优秀的 JavaScrip 框架 jQuery

第 7 章　Visual Studio 2010 新特性

Visual Studio 2010 开发环境包含了新的.NET Framework 版本和新的 C#语言版本，还添加了一种新的编程语言 F#，Visual Studio 集成开发环境也做了改进和完善，本章将介绍 Visual Studio 2010 在开发环境和语言方面的新特性。

7.1　集成开发环境的改进

Visual Studio 2010 与 Visual Studio 2008 开发环境相比在外观上有较明显的差别，在文本编辑、调试等各方面做了增加和改进。

7.1.1　新的窗口风格

Visual Studio 2010 使用 WPF 开发，其界面风格与以前的版本相比有明显的差别，Visual Studio 2010 旗舰版起始页界面如图 7.1 所示。



图 7.1　Visual Studio 2010 起始页

Visual Studio 2010 中的窗口具有更强的依靠功能，各个窗口（如代码窗口、设计窗口、调试窗口、输出窗口等）不但可以停靠在 Visual Studio 主窗口的任何一侧，而且还可以脱离 Visual Studio 主窗口成为一个独立的窗口，当然也可以方便地再附加到 Visual Studio 主窗口中。在任意一个窗口的标题栏双击，即可将其脱离 Visual Studio 主窗口成为独立窗口，如图 7.2 所示。

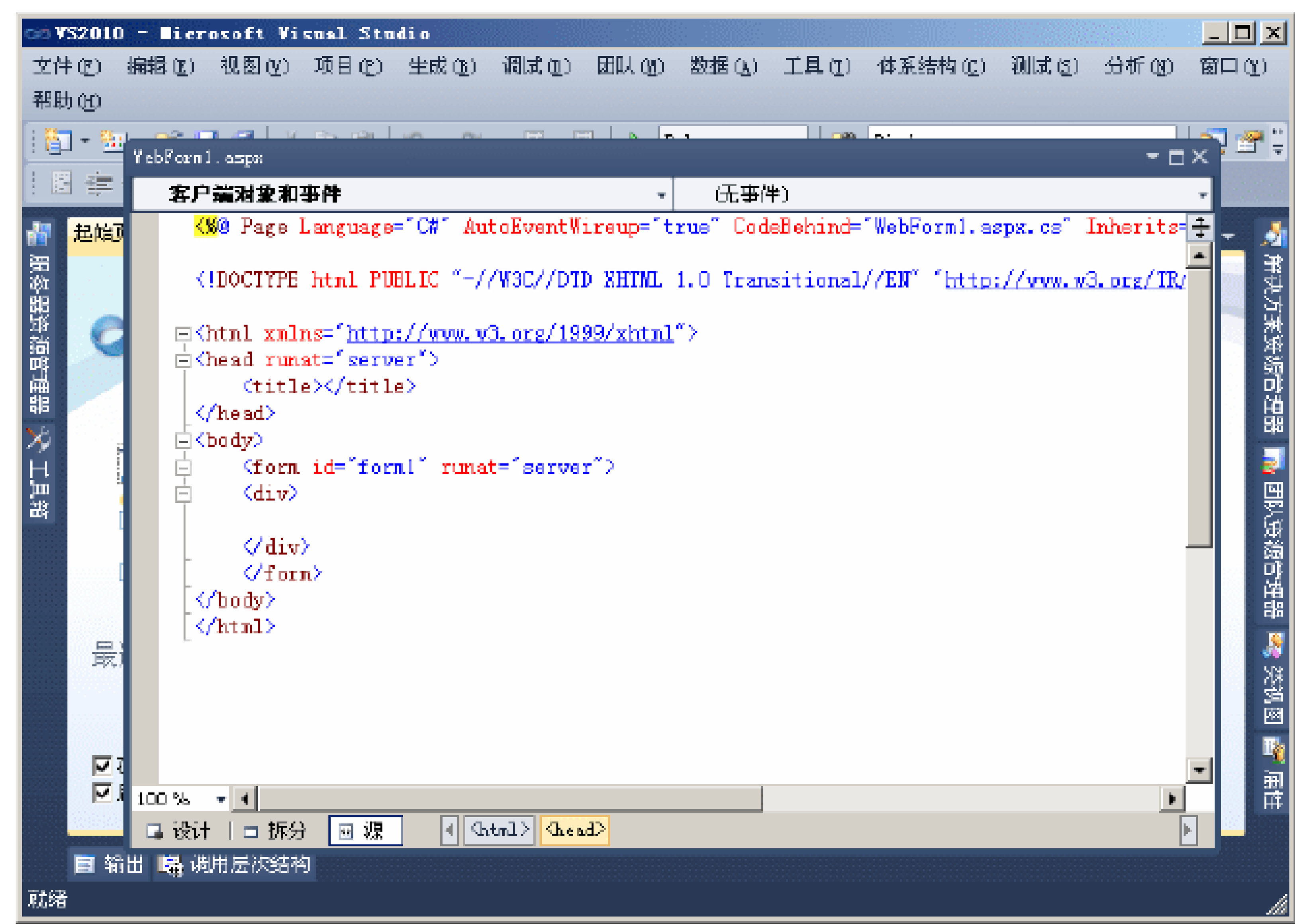


图 7.2　脱离 Visual Studio 主窗口的独立窗口

7.1.2　盒子选择和多行编辑

在 Visual Studio 2010 中，支持对代码的“盒子选择（Box Selection）”，即可以选择屏幕上任意一个矩形区域内的代码，而不要求是整行选择。进行盒子选择的操作方法是按住 Alt 键同时拖动鼠标，就可以选中鼠标所划过的矩形区域的代码，如图 7.3 所示。

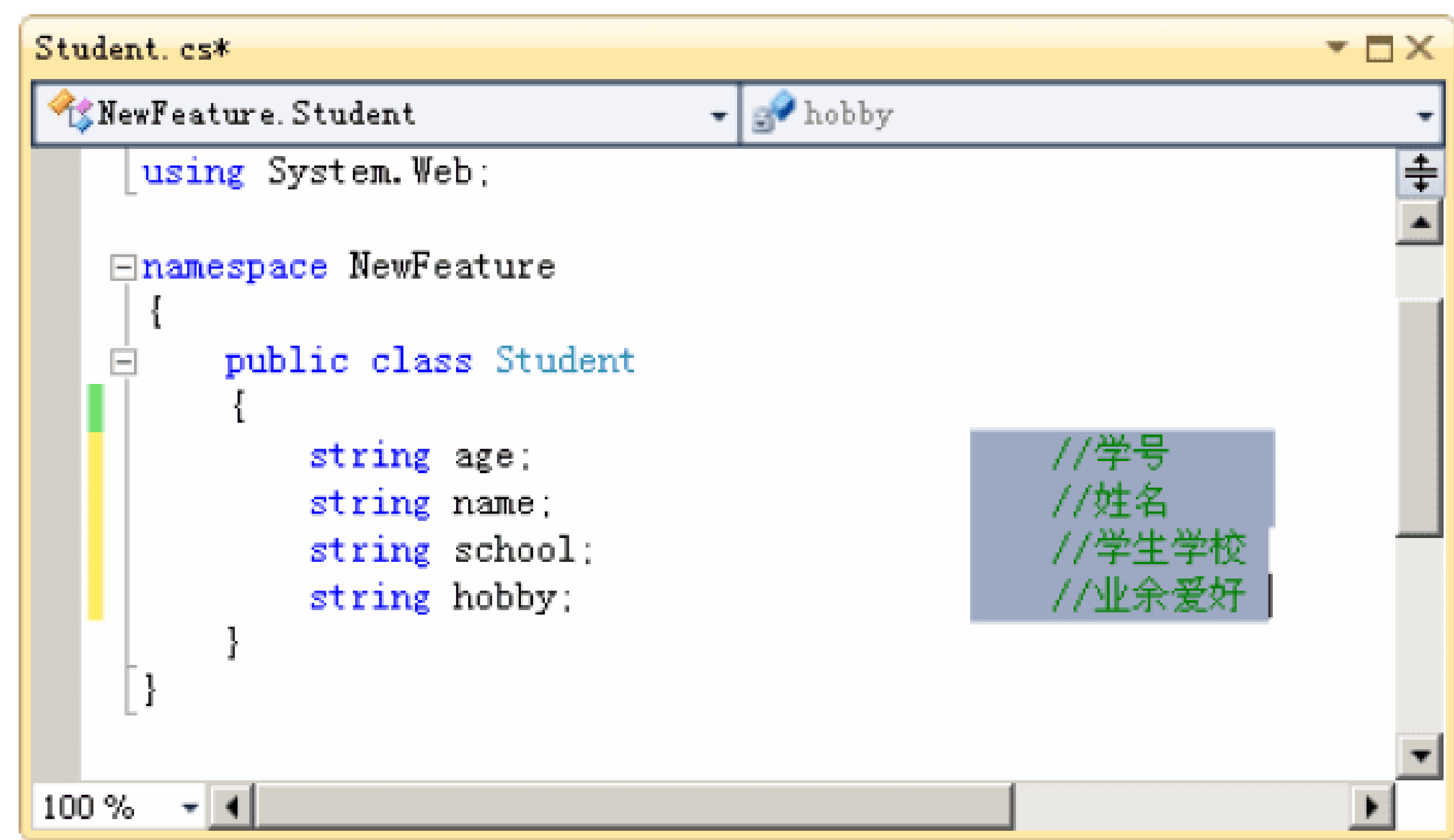


图 7.3　文本编辑器的盒子选择功能

使用盒子选择功能选中一个矩形区域后，如果从键盘进行输入，则所输入的内容会同时出现在所选择的所有行。利用这个特性，可以完成一些实用的功能。例如，在图 7.3 所示的 Student 类中，如果要将每一个字段的访问属性都变成 public，则可以用盒子选择方式选中各个 string 前面的空白，然后输入 public，就可以完成这个任务。

7.1.3　快速搜索

Visual Studio 2010 新增了一个快速搜索（Quick Search）功能，可以快速模糊搜索文件中的代码。在代码编辑器中同时按住 Ctrl 和，（逗号）键，就可以打开快速搜索窗口，在其中输入要查询的代码片断，即可以实时显示查找结果。例如，在某段代码中，程序员只记得某段代码包含 get 和 date，却忘记了精确的代码，则可以使用快速搜索窗口，在其中

输入 get+空格+date，就可以搜索出解决方案中同时包括 get 和 date 的代码，如图 7.4 所示。

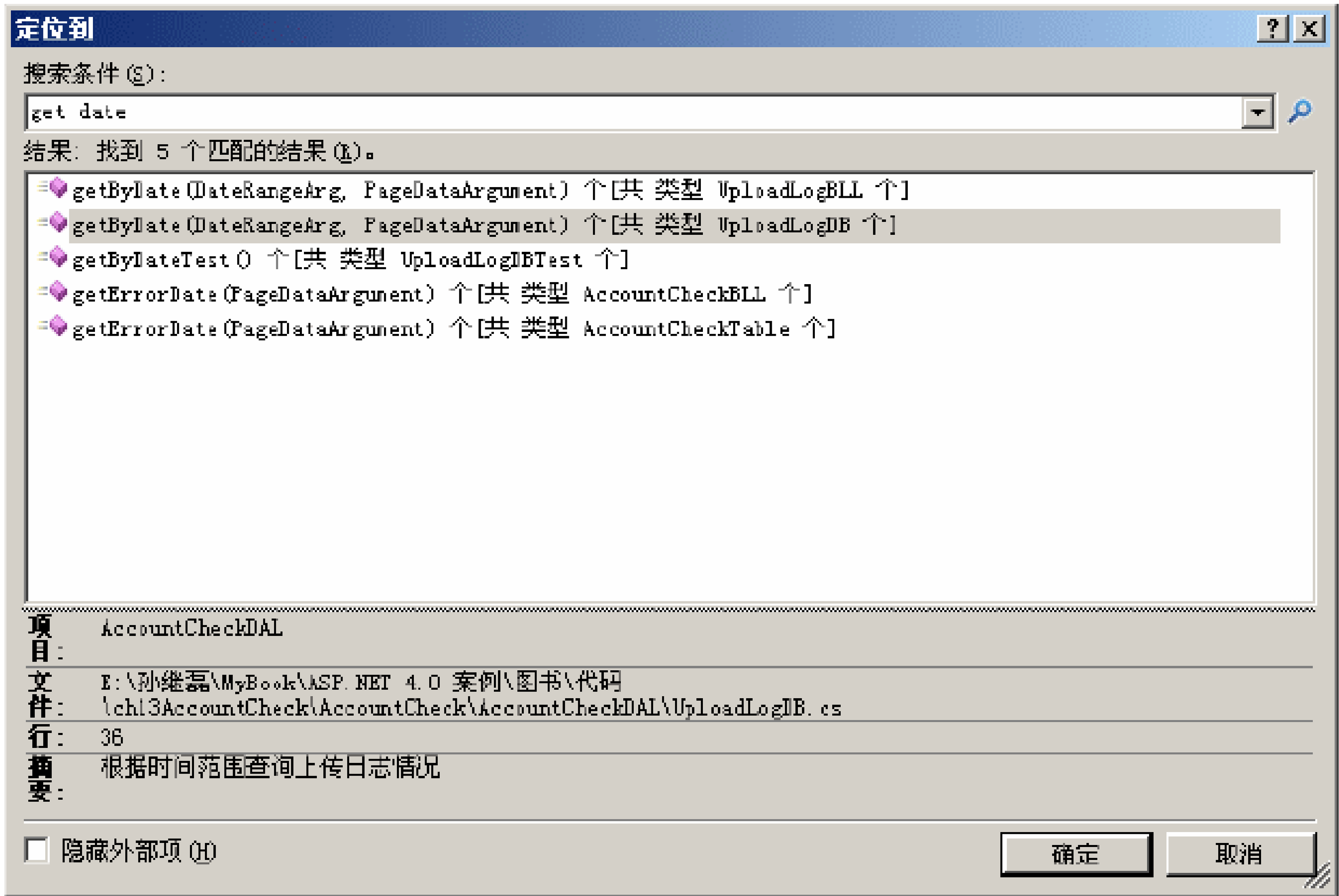


图 7.4　代码快速搜索

7.1.4　调用层次结构

在一个真实项目中，各个类的方法之间互相调用，形成一个复杂的调用关系网。Visual Studio 2010 新增了一个功能，能够显示方法之间的调用层次结构，帮助开发人员理清方法之间的调用关系。在代码视图中，在一个方法上右击，从弹出的菜单中选择“查看调用层次结构”选项，如图 7.5 所示。

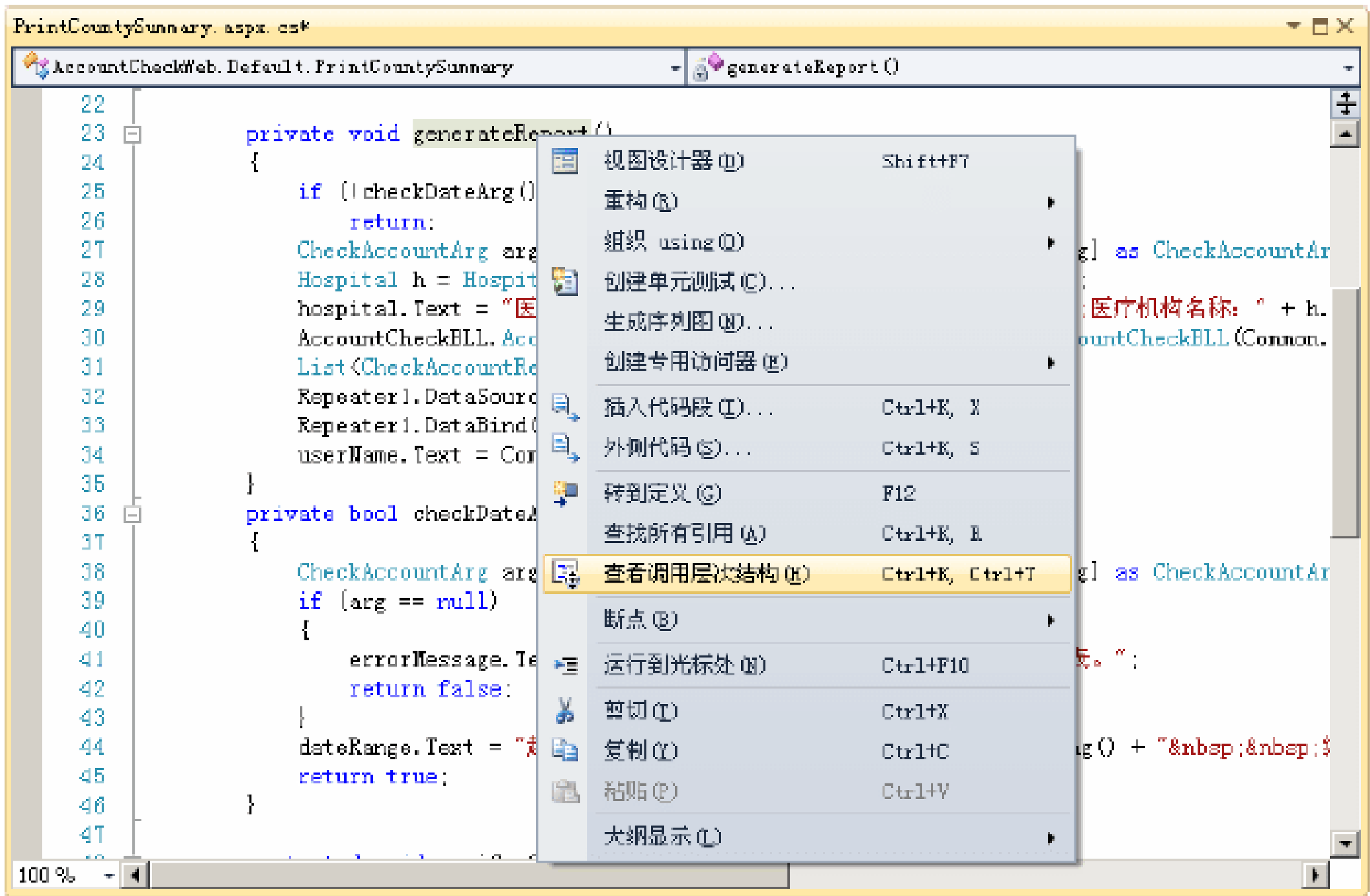


图 7.5　调用层次结构菜单

在图 7.5 中选择了“查看调用层次结构”命令后，则打开如图 7.6 所示的“调用层次结构”窗口。窗口左侧显示了调用所选择方法的方法和被所选择方法调用的方法。从左侧视图中选择一个方法，会在右侧视图中显示调用此方法的代码。

7.1.5　高亮显示引用

在 Visual Studio 2010 的代码编辑器中，把光标放在一个标识符上（不管是类、方法、

属性还是局部变量），就会高亮显示当前文件中对于所选标识符的所有引用，如图 7.7 所示，高亮显示了对局部变量 where 的所有引用。

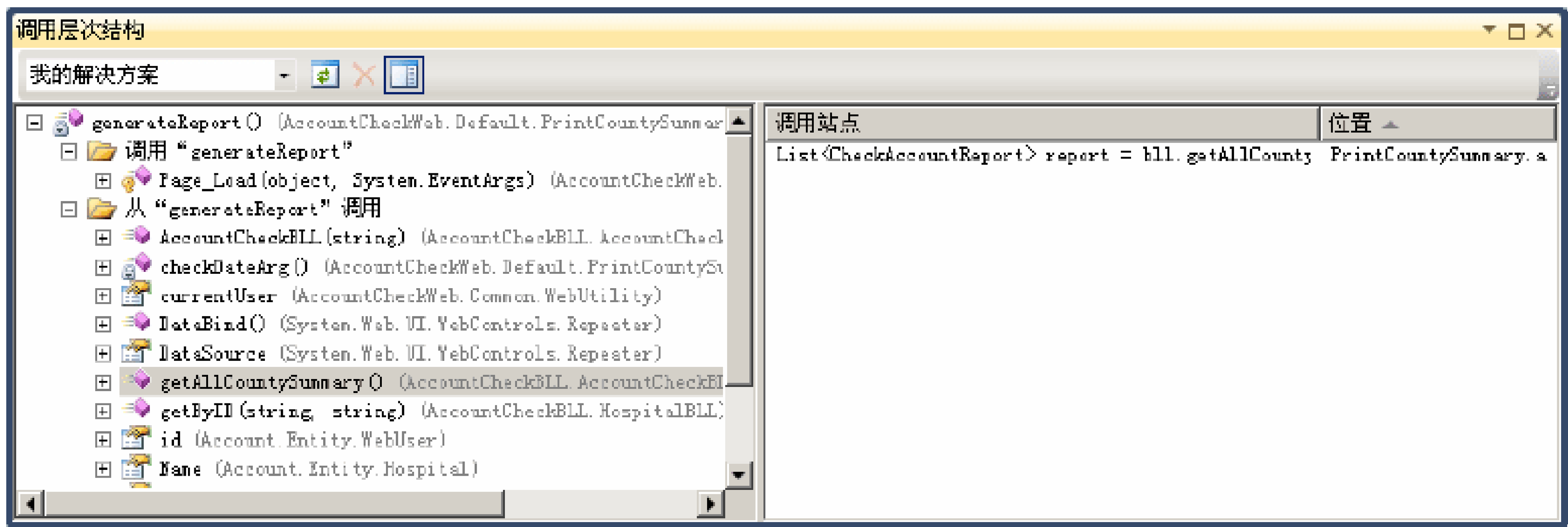


图 7.6　调用层次结构窗口

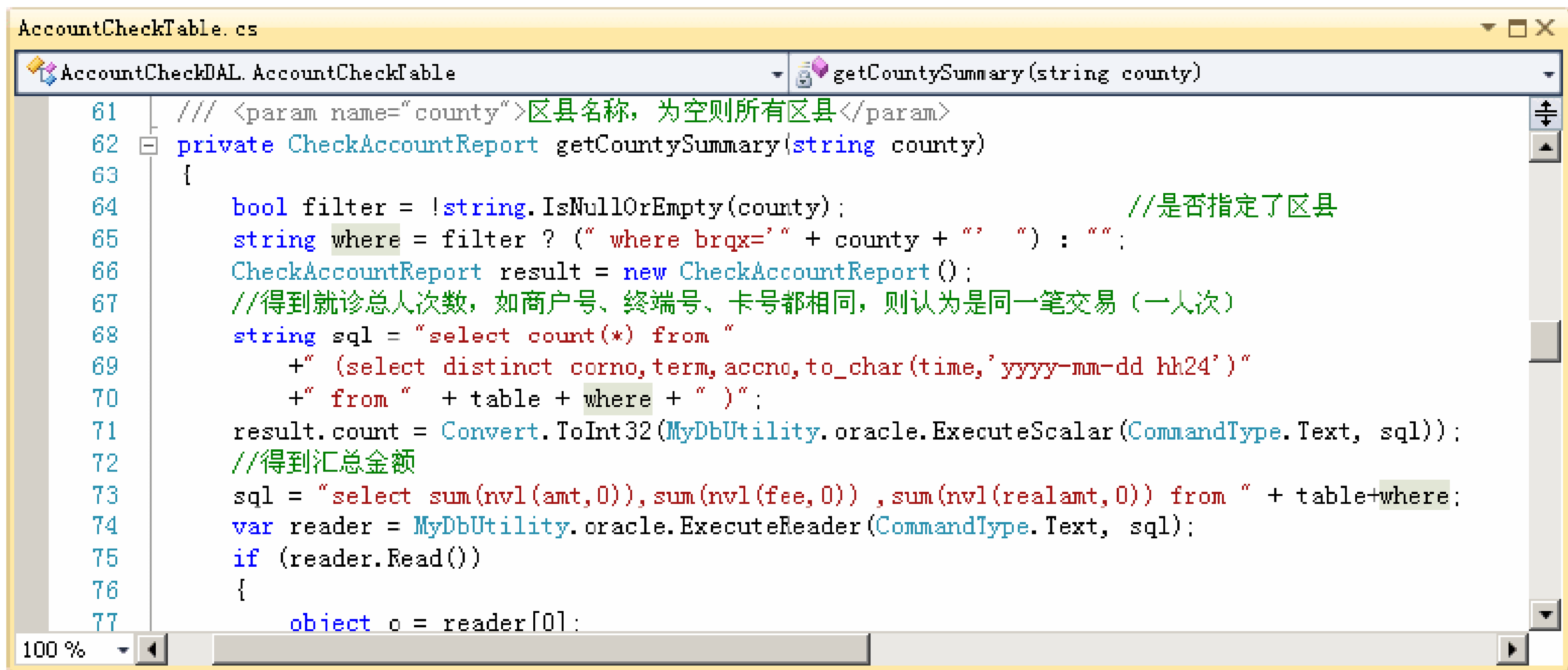


图 7.7　高亮显示引用

7.2　ASP.NET 4.0 新特性

Visual Studio 2010 中包含的 ASP.NET 版本为 4.0，ASP.NET 4.0 新增了控件静态 ID、图表控件、Web.config 转换等新特性。

7.2.1　控件静态 ID

在 ASP.NET 4.0 以前的版本中，ASP.NET 控件在最终生成的 HTML 代码中的 ID 由 ASP.NET 按照一定的算法动态生成，有时名字会很长，例如某页中一个 TextBox 控件在客户端生成的 ID 为“ContentPlaceHolder1_WebUserControl11_MyTextBox1”。开发人员无法控制 ID，也无法事先知道这个 ID。这种 ID 一方面使页面看起来代码较乱，控件名称不直观，另一方面也给浏览器端的 JavaScript 编写造成了不便。

在 ASP.NET 4.0 中，服务器端控件有一个 ClientIDMode 属性，如果将其设置为 Static，则控件在浏览器端的 ID 就是服务器端设置的 ID。下面通过一个例子来看控件静态 ID 与默认 ID 的区别。

【例 7-1】 控件静态 ID。

本例演示了 ASP.NET 服务器端控件默认的 ID 生成方式与静态 ID 的区别，以及静态

ID 对于客户端 JavaScript 编程的优势。

（1）创建一个 ASP.NET 空项目，在项目中添加一个母版页 Site1.master。

（2）在项目中添加一个用户控件 WebUserControl1.ascx，在其中添加两个 TextBox 控件，一个使用默认 ID 生成方式，另外一个使用静态 ID，代码如下：

```
<asp:TextBox runat="server" id="TextBox1"> </asp:TextBox>
<asp:TextBox runat="server" id="TextBox2" ClientIDMode="Static" >
</asp:TextBox>
```

（3）从母版页 Site1.master 创建一个内容页面 WebForm2.aspx，在页面上放置一个刚才所创建的用户控件 WebUserControl1，然后再放置两个 Button 控件，一个使用默认 ID 生成方式，另外一个使用静态 ID，代码如下：

```
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
    <uc1:WebUserControl1 ID="WebUserControl11" runat="server" /><br />
    <asp:Button runat="server" id="button1" Text="默认 ID" />
    <asp:Button runat="server" id="button2" ClientIDMode="static" Text="
    静态 ID" />
</asp:Content>
```

（4）在浏览器中浏览 WebForm2.aspx 页面，并查看其 HTML 源代码，可以得到页面最终生成的 HTML，代码如下：

```
<div>
<input name="ctl00$ContentPlaceHolder1$WebUserControl11$TextBox1" type=
"text" id="ContentPlaceHolder1_WebUserControl11_TextBox1" />
<input name="ctl00$ContentPlaceHolder1$WebUserControl11$TextBox2" type=
"text" id="TextBox2" />
<br />
<input type="submit" name="ctl00$ContentPlaceHolder1$button1" value="默认
ID" id="ContentPlaceHolder1_button1" />
<input type="submit" name="ctl00$ContentPlaceHolder1$button2" value="静态
ID" id="button2" />
</div>
```

从上述 HTML 代码可以看出，默认情况下，ASP.NET 服务器端控件生成了较长的客户端 ID，而使用静态 ID 的控件其客户端 ID 与服务器端 ID 相同。对照 HTML 代码与 ASP.NET 页面代码还可以看出，默认情况下，控件的客户端 ID 值为其各级容器控件的 ID 与控件本身 ID 的组合。例如，内容页中 Button1 控件的容器为 ContentPlaceHolder1 控件，故 Button1 的客户端 ID 为 ContentPlaceHolder1_button1。页面上 TextBox1 的容器为用户控件 WebUserControl1，而 WebUserControl1 的容器为 ContentPlaceHolder1，故 TextBox1 控件的客户端 ID 为 ContentPlaceHolder1_WebUserControl11_TextBox1。

（5）如果要在浏览器端使用 JavaScript 操作两个 TextBox 控件，例如获取其中的文本，实现难度和工作量都有较大差别。对于静态 ID 控件来说，由于可以明确知道其客户端 ID，因此 JavaScript 编写很简单，只需使用以下代码即可。

```
function getValue1() {
    var v = document.getElementById('TextBox2').value;
    alert(v);
}
```

（6）要获得页面中采用默认 ID 命令的 TextBox 控件的值，则需要采用较为复杂的方法。在 JavaScript 中获得 ASP.NET 服务器控件 ID 通常使用以下代码。

```
var id = '<%=button1.ClientID%>';        //其中 button1 为 ASP.NET 服务器控件
```

在本例中，由于 TextBox1 位于用户控件中，是一个 private 成员，所以不能直接采用上述代码获得其 ID。可以在用户控件中添加一个 public 属性，通过此属性返回 TextBox 控件的客户端 ID，代码如下：

```
public string textBoxId
{
    get { return TextBox1.ClientID; }
}
```

在 WebForm2.aspx 页面中，通过访问此属性得到 TextBox 控件的客户端 ID，并应用于 JavaScript 代码中。

```
function getValue2() {
    var v = document.getElementById('<%=WebUserControl11.textBoxId%>').
    value;
    alert(v);
}
```

7.2.2　图表控件

ASP.NET 4.0 中的图表控件（Chart 控件）类似于 Excel 中的图表控件，可以根据数据源生成各种常见图形报表，如条形图、饼形图等，图 7.8 为 Chart 控件的生成的两个图表。

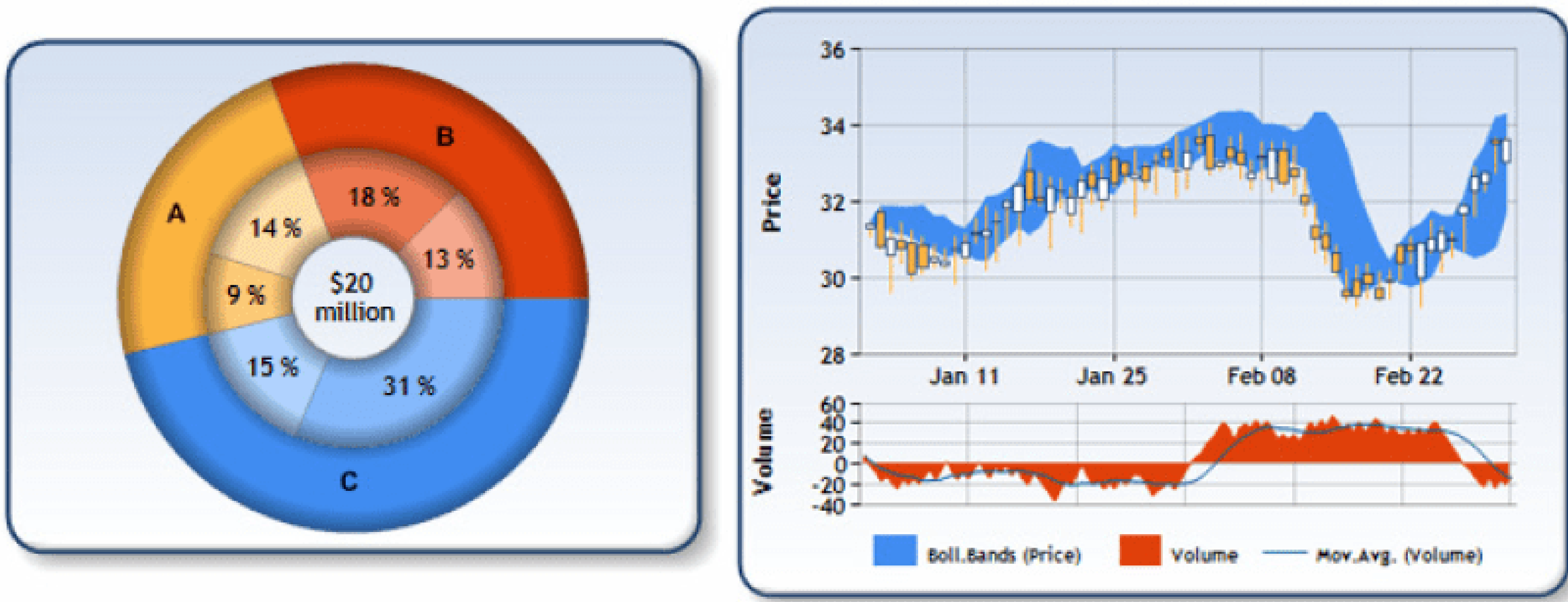


图 7.8　Chart 控件示例

【例 7-2】　图表控件示例。

本例演示了图表控件的简单使用。

（1）创建一个 ASP.NET Web 应用程序，添加一个页面 ChartSample.aspx。

（2）打开 ChartSample.aspx 页面，从工具箱的“数据”控件组中找到“Chart”控件，如图 7.9 所示，将其拖动到页面上。

（3）在 ChartSample.aspx 页面中选中 Chart 控件，单击其右上角的按钮，从打开的智能任务页面中选择“新建数据源”选项，如图 7.10 所示。

（4）在弹出的“数据源配置向导”对话框中选择“SQL 数据库”，如图 7.11 所示。

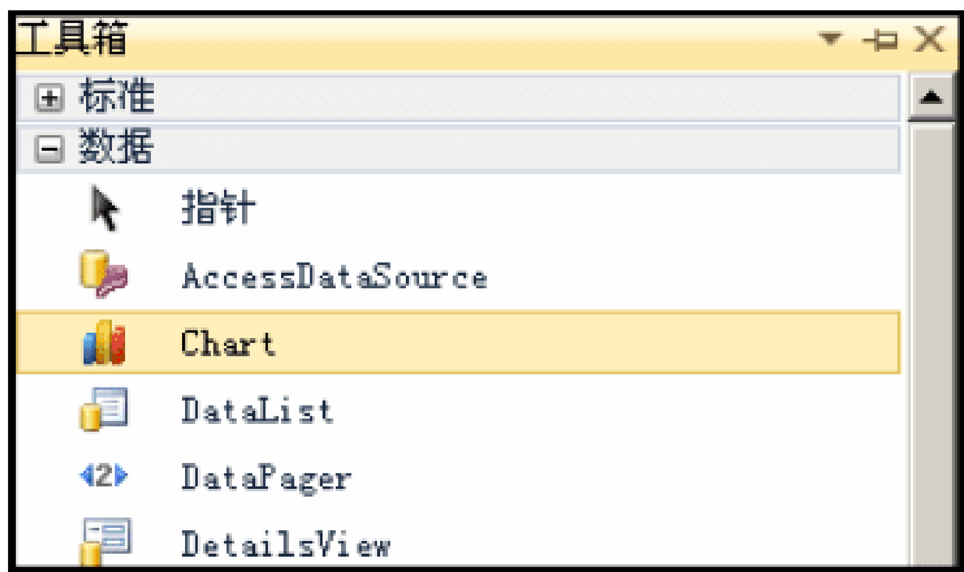


图 7.9　工具箱里的 Chart 控件

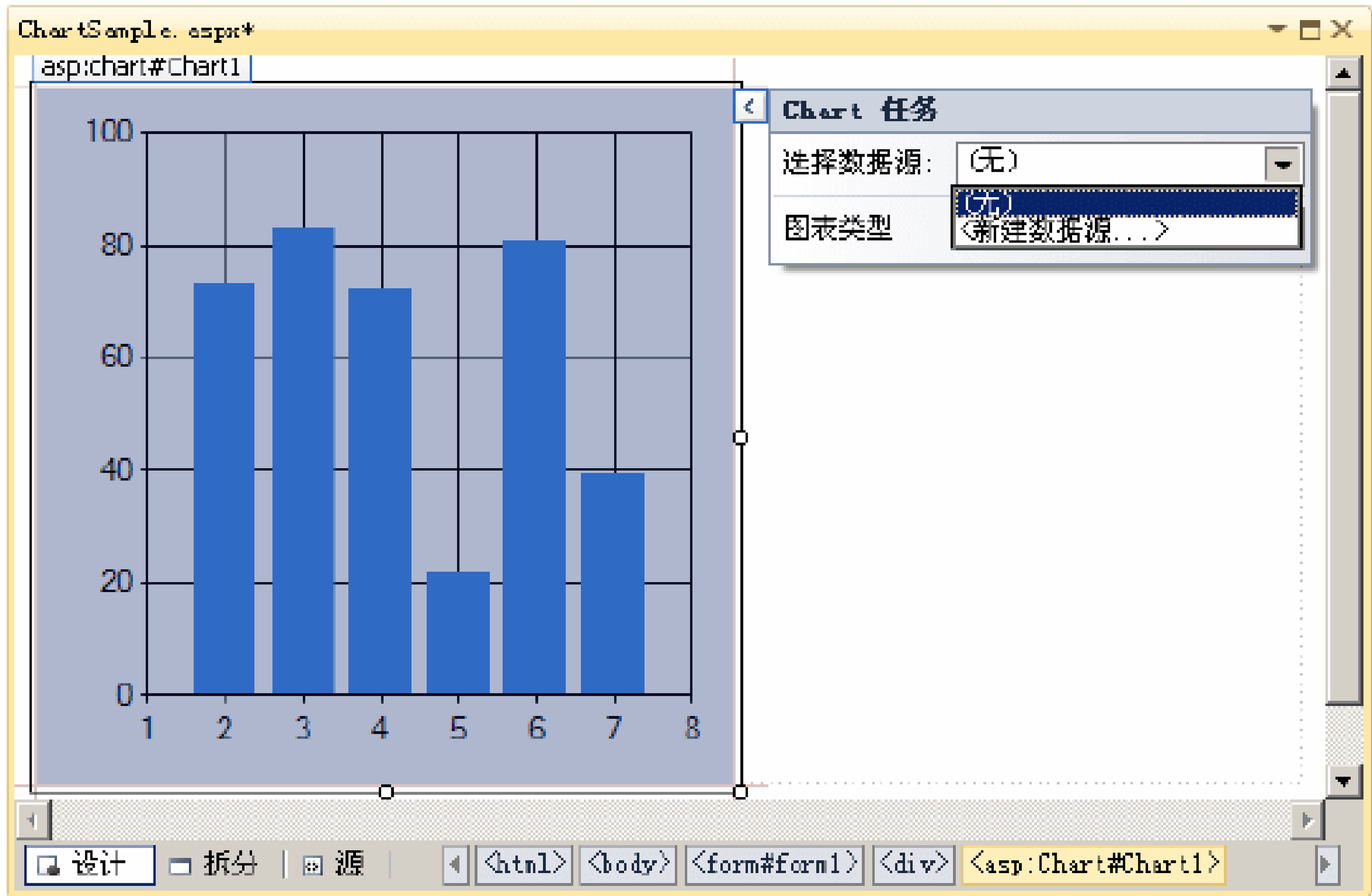


图 7.10　为 Chart 控件添加数据源

（5）在“数据源配置向导”的后续步骤中，设置服务器为本机，数据库为 Northwind，查询语句为“select CategoryName, CategorySales from [Category Sales for 1997]”，其中 Category Sales for 1997 是 Northwind 数据库中一个视图，其中数据为 1997 年按照产品类别统计的销售额。

（6）在 Chart 控件的智能任务面板中，设置其横坐标和纵坐标使用的数据来源，其中横坐标为产品类别名称 CategoryName，纵坐标为销售金额，如图 7.12 所示。

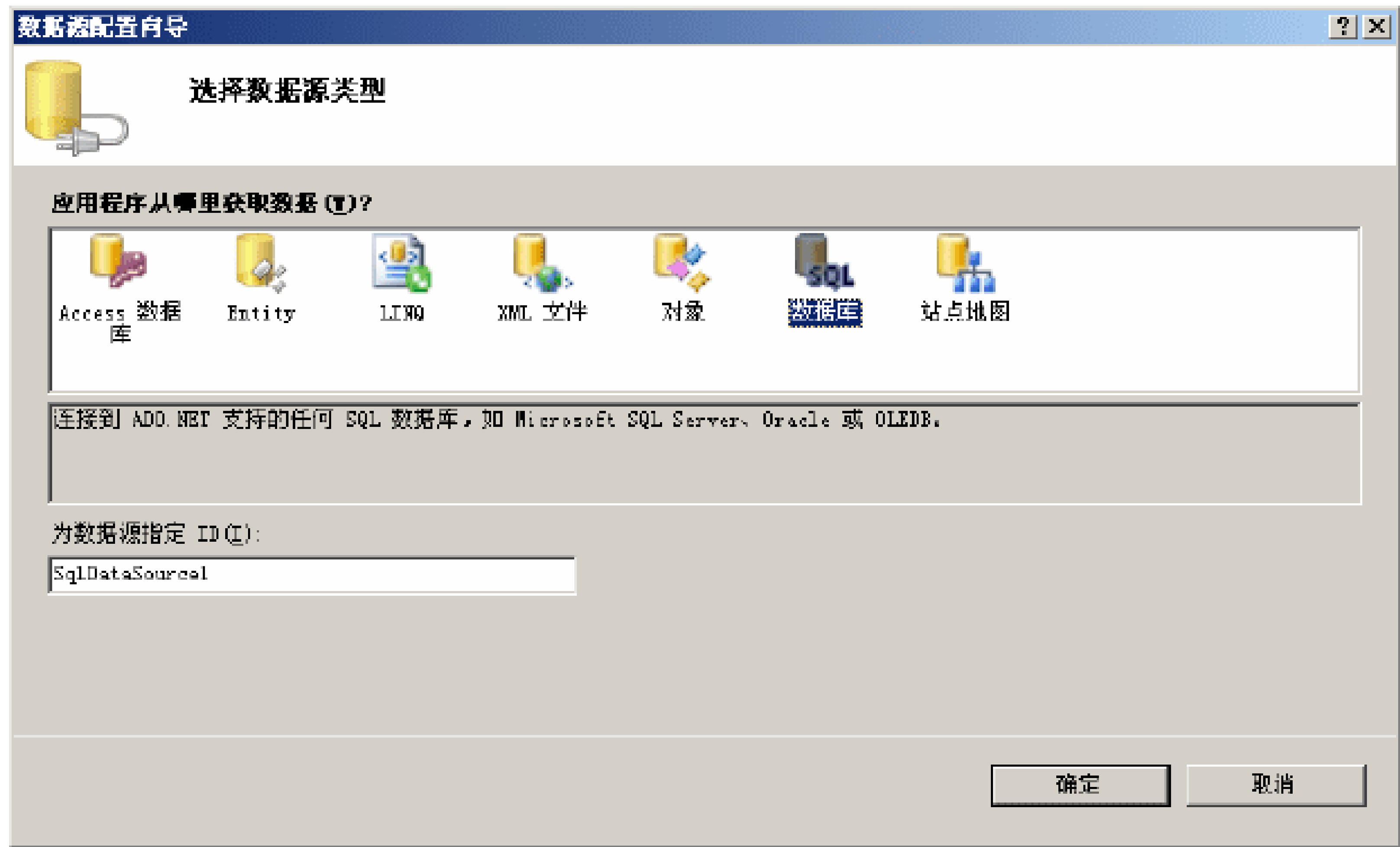


图 7.11　配置数据源

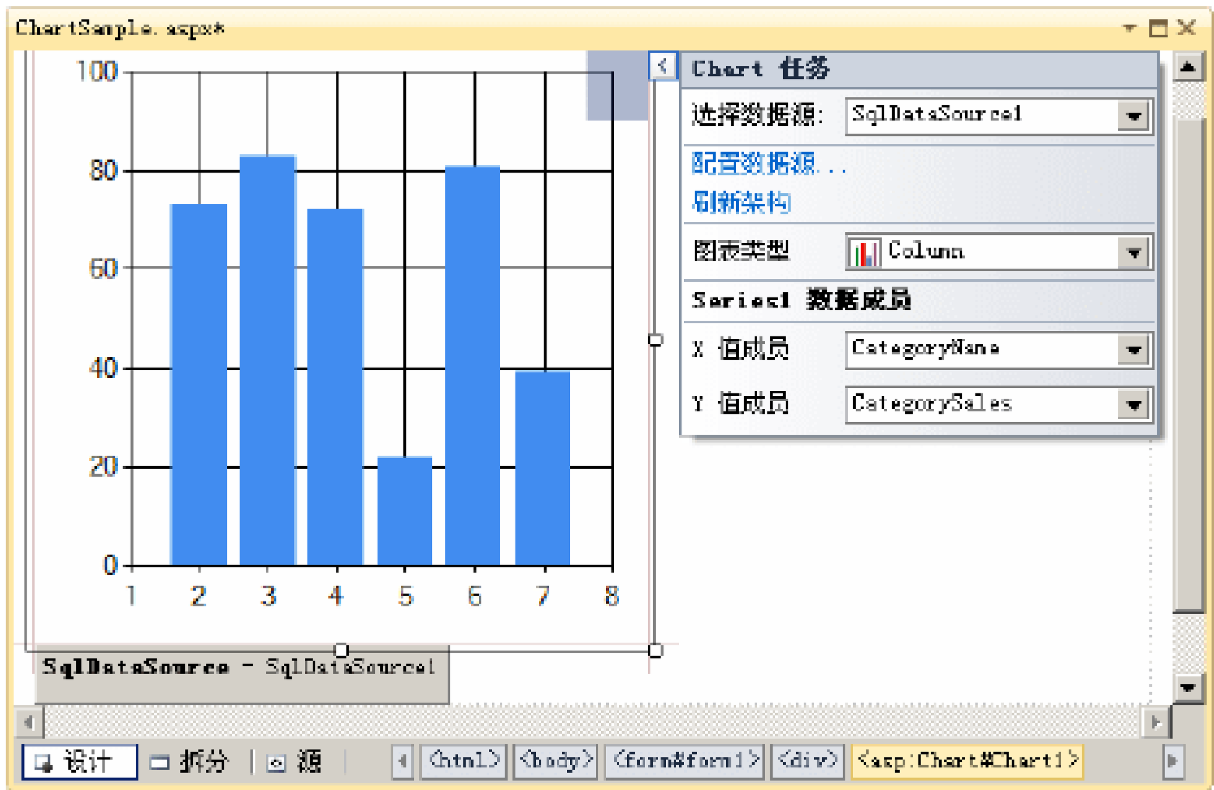


图 7.12　设置坐标数据

（7）在浏览器中查看 ChartSample.aspx 页面，运行界面如图 7.13 所示。

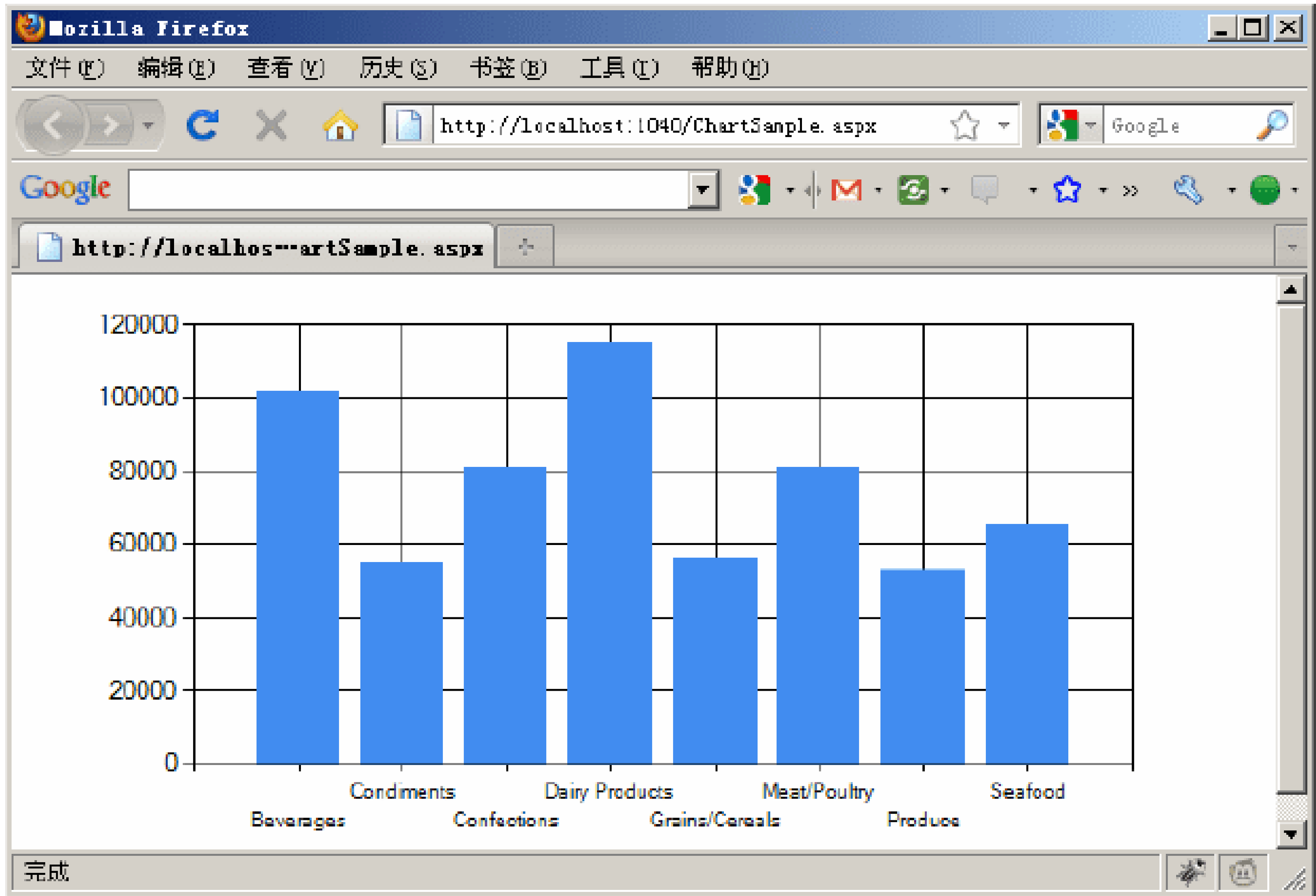


图 7.13　ChartSample 运行界面

7.2.3　Web 配置文件转换

ASP.NET 应用程序的配置文件 Web.config 中通常包含一些环境参数，如连接字符串等。当 ASP.NET 应用程序运行环境发生改变时，Web.config 里面的内容通常也需要进行调整。例如，在开发环境与部署环境下，数据库连接字符串不同，自定义错误也不相同。在 ASP.NET 4.0 以前，需要通过手工修改 Web.config 文件内容的方式实现这个功能。在 ASP.NET 4.0 中，引入了一个新功能 Web 配置文件转换（Web.config transformation）可以自动完成这个功能。

Web 配置文件转换的原理为在 ASP.NET 项目中定义一个基本的 Web 配置文件，再定义若干个转换文件，每个转换文件对应一种特定的配置环境，最常见的有两种环境，分别是调试环境（Debug）和部署环境（Release）。在转换文件中定义一些转换规则，对基本 Web 配置文件中的内容进行转换（如删除、替换等），从而得到一个新的 Web 配置文件。当生成 ASP.NET 项目时，选择一个合适的配置环境（如 Debug、Release 等），就可以得到适应于此环境的 Web.config 文件。

【例 7-3】 Web 配置文件转换。

本例演示如何利用 Web 配置文件转换功能生成不同环境下的 Web 配置文件。

（1）创建一个 ASP.NET Web 应用程序。

（2）从菜单中选择“生成”|“配置管理器”命令，则打开“配置管理器”窗口，如图 7.14 所示。

（3）在图 7.14 所示的配置管理器中，可以看到目前已经存在两个配置：Debug 和 Release，这两个配置是 ASP.NET 自动生成的。从顶部的下拉列表框中选择“新建”选项，则弹出如图 7.15 所示的“新建解决方案配置”对话框，在其中输入名称 MyConfig，并单击“确定”按钮。

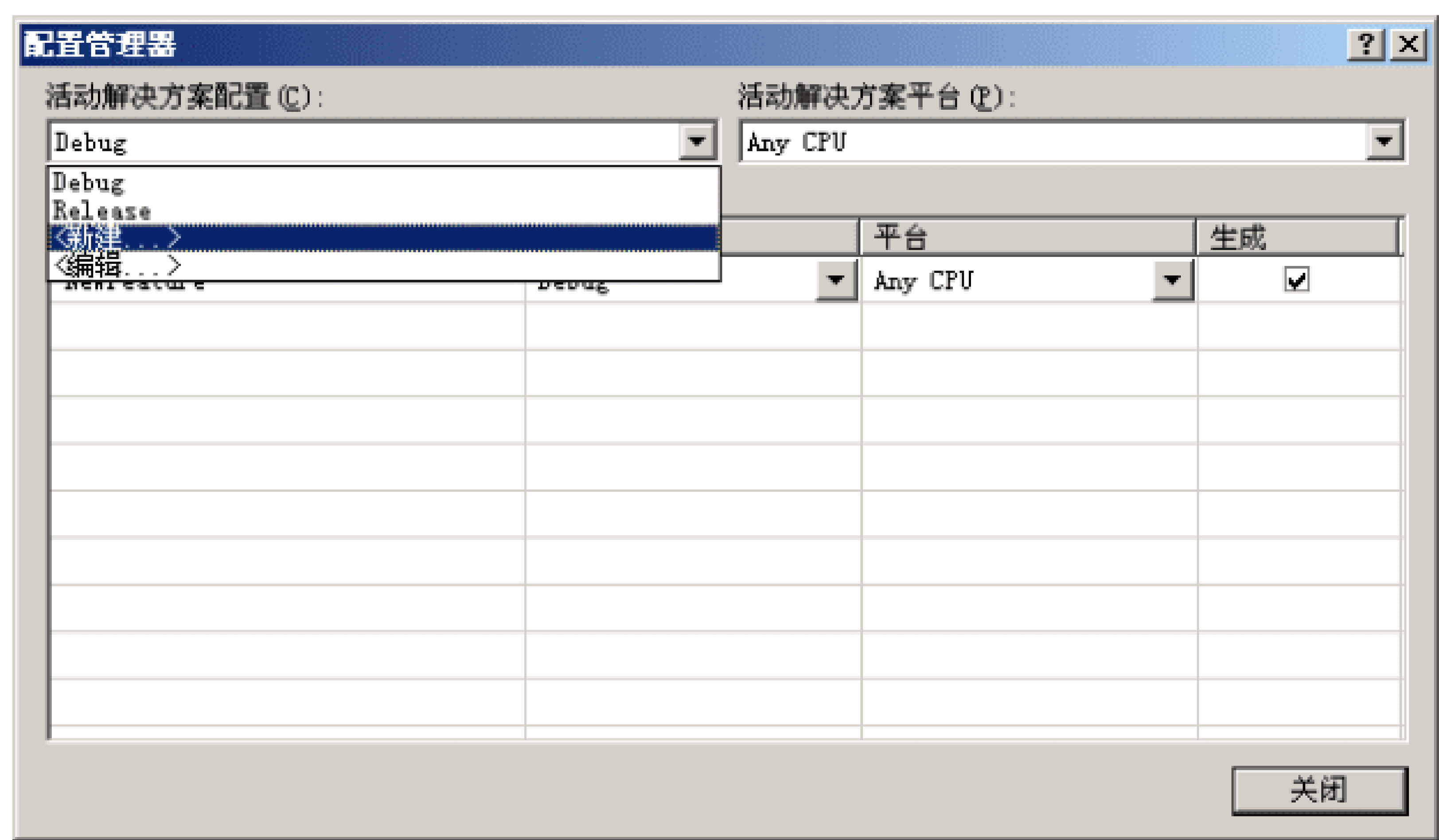


图 7.14　配置管理器

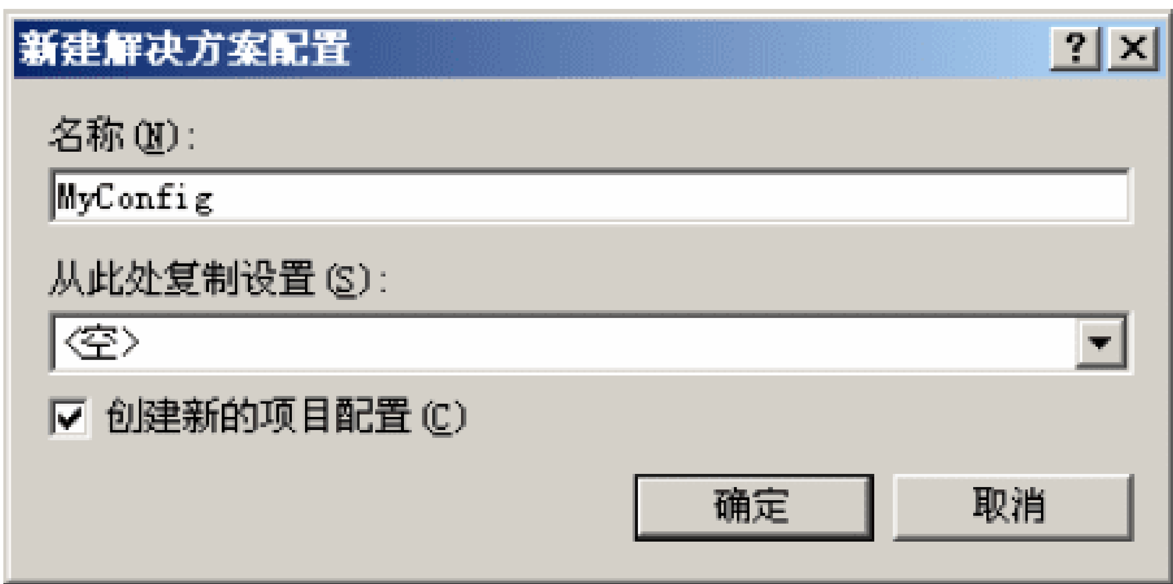


图 7.15　新建解决方案配置

（4）在解决方案资源管理器中，右击 Web.config 文件，从弹出的菜单中选择“添加配置转换”命令，则会在解决方案中添加一个新的文件 Web.MyConfig.config，其中 MyConfig 为刚才所添加的解决方案配置名称，如图 7.16 所示。

（5）为了验证配置文件转换功能，在项目中分别为默认的配置文件 web.config 和各个配置转换文件添加不同的内容。首先打开默认配置文件 Web.config，在其中输入两个连接字符串，代码如下：

```
<connectionStrings>
    <add  name="db1"  connectionString="Data  Source=.;  Initial  Catalog=
    DefaultDb; Integrated Security=True"/>
    <add  name="db2"  connectionString="Data  Source=.;  Initial  Catalog=
    DefaultDb; Integrated Security=True"/>
</connectionStrings>
```

（6）打开 Web.Debug.config 文件，这是一个配置转换文件，其中包含了一些配置文件替换规则。在 Web.Debug.config 文件中输入以下内容。

```
<connectionStrings>
  <add name="db1"
    connectionString="Data Source=DebugServer; Initial Catalog=Northwind;
    Integrated Security=True"
    xdt:Transform="SetAttributes" xdt:Locator="Match(name)"/>
</connectionStrings>
```

上述代码定义了对 ConnectionString 的一个转换规则，xdt:Transform="SetAttributes"表示转换方式为设置属性，xdt:Locator="Match(name)"表示匹配规则为名称相同。整个代码的含义为查找 name 值相同的连接字符串，并用此处提供的新值设置原有属性。

提示：ASP.NET 支持的配置文件转换方式有多种，除了 SetAttributes 外，还有 Replace、Remove、Insert 等，分别表示替换、删除和插入，详情可参见 MSDN。

（7）打开 Web.MyConfig.config 配置转换文件，在其中输入以下内容：

```
<connectionStrings>
  <add name="db1"
    connectionString="Data Source=MyServer; Initial Catalog=Northwind;
    Integrated Security=True"
    xdt:Transform="SetAttributes" xdt:Locator="Match(name)"/>
</connectionStrings>
```

（8）从 Visual Studio 工具栏中选中 Debug 作为当前配置，然后从菜单中选择“生成”|“发布”命令，则弹出如图 7.17 所示的“发布 Web”对话框。在对话框的“发布方法”下拉列表框中选择“文件系统”，并在“目标位置”下拉列表框中设置一个合适的发布路径，然后单击“发布”按钮以发布项目。

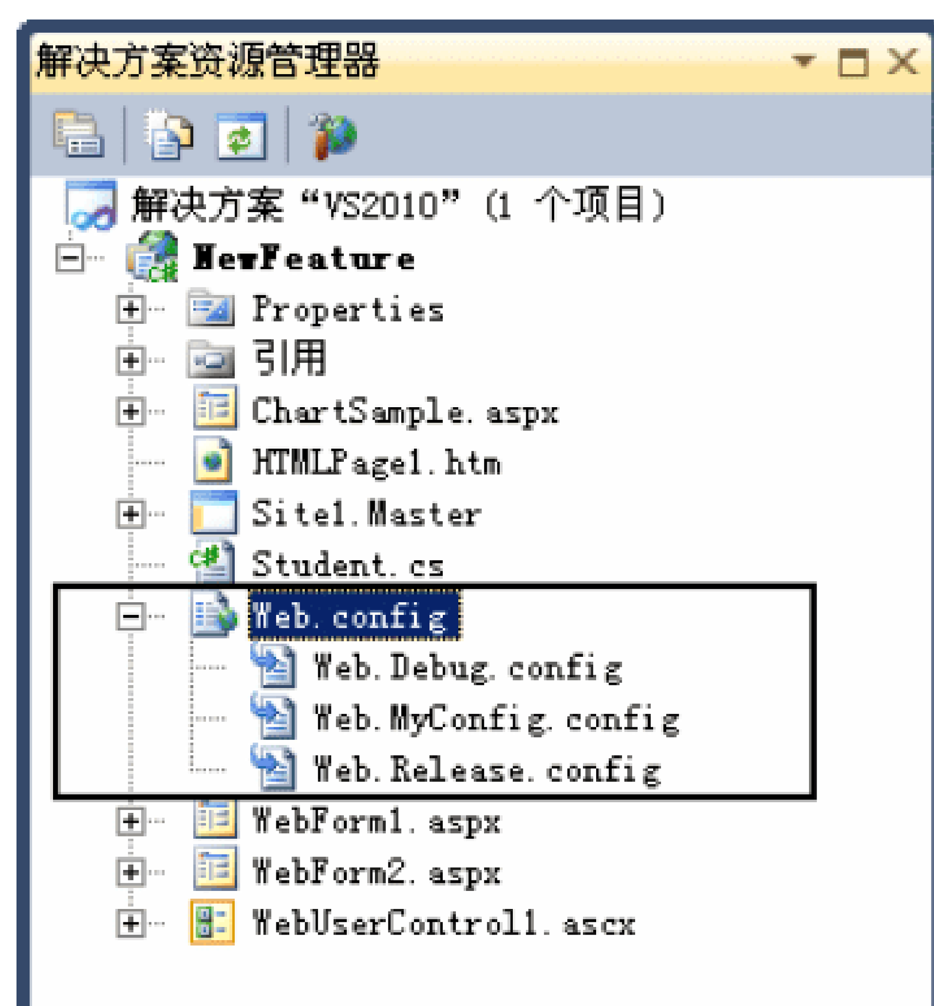


图 7.16　新添加的配置转换

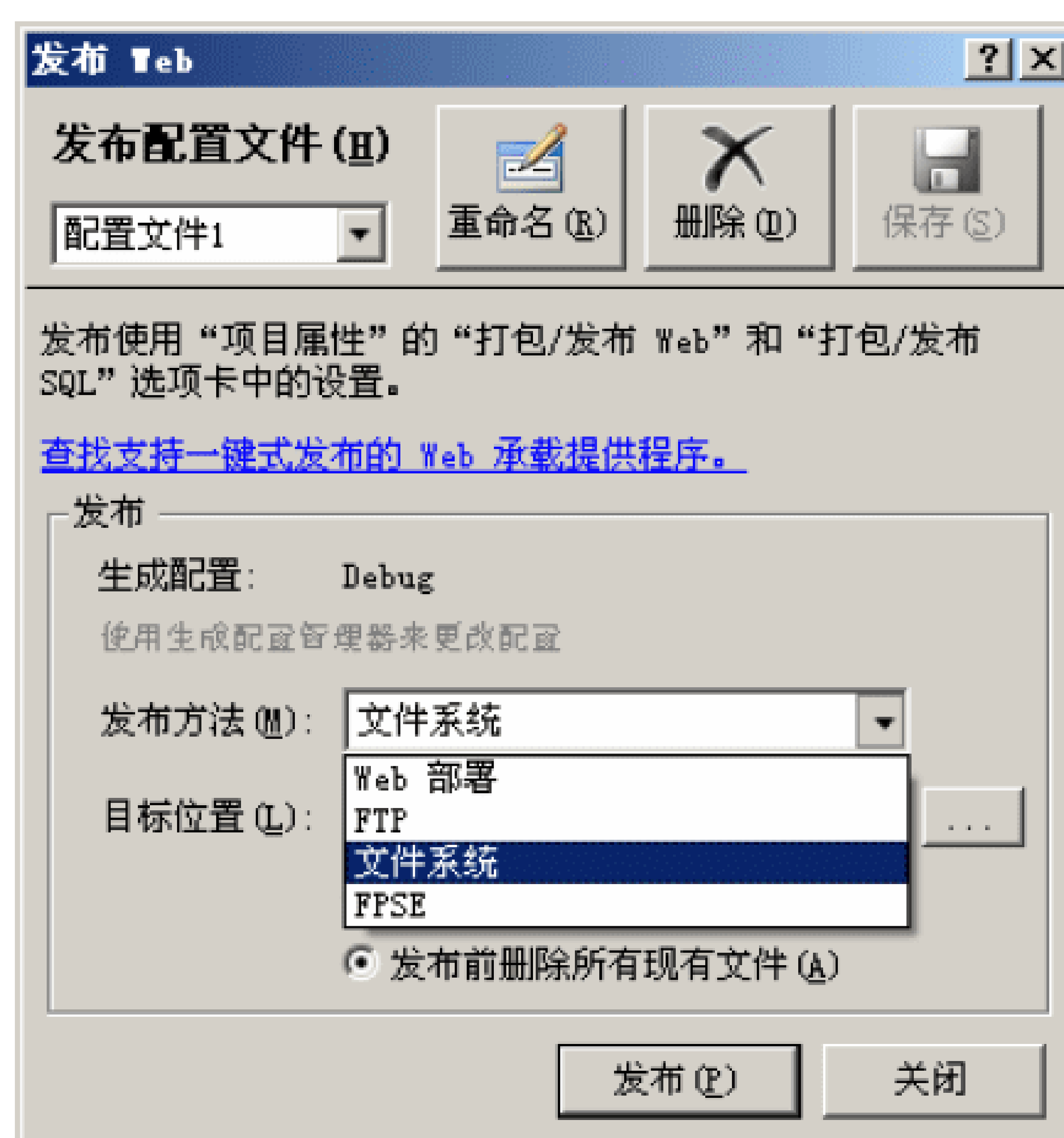


图 7.17　“发布 Web”对话框

（9）发布成功后，找到发布的目标位置，打开 Web.config 文件，可以看到连接字符串内容如下：

```
<connectionStrings>
  <add name="db1" connectionString="Data Source=DebugServer; Initial
  Catalog=Northwind; Integrated Security=True"/>
  <add name="db2" connectionString="Data Source=.; Initial Catalog=
  DefaultDb; Integrated Security=True"/>
</connectionStrings>
```

对照以上连接字符串内容和开发环境中的各个连接字符串可以看出，开发环境中 Web.config 文件中第 1 个连接字符串被替换为 Web.Debug.config 的内容，而第 2 个连接字符串未发生改变。

（10）在 Visual Studio 开发环境中选中 MyConfig 作为当前配置，再次发布网站，并查看发布后的 Web.config 文件，得到以下内容。

```
<connectionStrings>
  <add name="db1" connectionString="Data Source=MyServer; Initial Catalog=
  Northwind; Integrated Security=True"/>
  <add name="db2" connectionString="Data Source=.;Initial Catalog=
  DefaultDb; Integrated Security=True"/>
</connectionStrings>
```

7.3　C# 4.0 新特性

在 Visual Studio 2010 中，C#语言升级到最新版本 4.0。C# 4.0 中增加了动态类型、命名和可选参数、协变逆变等新功能。

7.3.1　动态类型

在 C# 4.0 以前的版本中，变量都是静态类型，即变量的类型在编译阶段就能够确定。即使使用 var 关键字声明的变量，在声明的同时必须提供一个默认值，根据这个值就可以确定出变量的类型，因此这个变量实际上也是静态变量。

所谓动态类型，是指无法在编译阶段确定变量类型，只有等到运行时才能获得变量实际类型。动态类型的好处是能够编写更加灵活、强大、简洁的代码。其劣势在于：（1）由于编译阶段无法知道变量的类型，因此不进行类型安全检查；（2）性能要比静态类型低；（3）无法在 Visual Studio 开发环境中显示智能提示。由于以上原因，除非必要，否则不要使用动态类型。

C# 4.0 中使用 dynamic 关键字定义动态类型。动态类型的意味着写代码时不能确定这个类型是什么，也不知道这个类型有什么方法和属性。对动态类型可以执行任意操作：赋值或调用任何方法、访问任何属性、作为参数传递给函数等。如下面的代码所示。

```
dynamic myDynamic = getDynamic();            //此方法返回一个不能确定类型的对象
myDynamic.anyMethod("Hello");                //调用方法
myDynamic.property1 = myDynamic.property2;   //访问属性
int Quux = myDynamic[0];                     //索引器
```

```
int Qux = myDynamic(12,5);                     //作为方法（委托）使用
someMethod(myDynamic);                         //作为方法参数传递
```

由于 myDynamic 是动态类型，在编译时无法知道其类型、属性、方法，无法进行类型安全检查，所以以上代码会编译通过。如果 myDynamic 不支持某个属性或者方法，或者其类型与函数所期待的参数类型不一致，只有等到程序运行时才会发现这个错误。

7.3.2　命名和可选参数

在开发项目过程中，有时需要编写一个接受多个方法的参数，例如，某销售管理系统需要根据一系列条件查询销售情况，过滤条件为销售日期、商品类别、商品编号、销售城市的任意组合，如果指定了某个条件，则根据这个条件进行过滤，否则不对此条件进行限制。如果用一个方法实现此功能，则此方法需要接受上述所有查询条件作为参数，代码如下：

```
/// <summary>
/// 根据用户指定条件查询销售情况
/// </summary>
/// <param name="from">查询开始时间</param>
/// <param name="to">查询截止时间</param>
/// <param name="categoryId">商品类别 ID</param>
/// <param name="productId">商品 ID</param>
/// <param name="cityId">销售城市 ID</param>
/// <returns>查询得到的销售记录列表</returns>
static List<SaleRecord> select( DateTime from, DateTime to, string
categoryId,
    string productId, string cityId)
{ ... }
```

在调用上述方法执行查询时，很多情况下并不需要根据所有条件进行查询，而只需要根据部分条件进行查询。在 C# 4.0 以前，如果要调用这个方法，必须为所有参数都指定一个值，即使不需要根据这个条件进行查询。例如，如果要根据销售日期和销售城市进行查询，则需要使用以下代码。

```
var list = select(DateTime.Parse("2010-1-1"), DateTime.Parse("2010-1-10"),
null, null, "1001");
```

在 C# 4.0 中，允许为方法的参数指定默认值，则此参数成为可选参数，调用方法时可以不必给这个参数传值。C# 4.0 还支持命名参数，调用方法时可以通过参数名称为某个特定参数传值。

提示：可选参数必须位于方法参数列表的最后，即在可选参数后面不允许出现不可选参数。

利用 C# 4.0 的可选参数，上述代码可以修改成以下形式。

```
static List<SaleRecord> select( DateTime from, DateTime to, string
categoryId=null,
   string productId=null, string cityId=null)
{ ... }
```

如果要查询某个时间段的销售情况，则可以使用以下代码。

```
var list = select(DateTime.Parse("2010-1-1"), DateTime.Parse("2010-1
-10"));
```

如果要查询某个时间段某城市的销售情况，则可以使用以下代码。

```
var list = select(DateTime.Parse("2010-1-1"), DateTime.Parse("2010-1-10"),
cityId:"1010");
```

上述代码中的 cityId:"1010"表示为方法的 cityId 参数传递的值为 1010，这就是 C# 4.0 使用命名参数的语法。

7.3.3　协变和逆变

在 C# 4.0 中引入了协变（Covariance）和逆变（Contravariance）的概念以增强泛型接口和委托。在 C#中，将一个 IList<string>类型转换为 IList<object>是不允许的，如下代码在编译时会出错。

```
IList<string> words = new List<string> { "this", "is", "a", "string",
"array" };
IList<object> objects = words;
```

上述代码编译时会提示“无法将类型 IList<string>隐式转换类 IList<object>”，如果确实要实现转换，则需要一个强制类型转换。有些刚接触 C#的程序员可能认为 String 可以隐式转换为 Object，所以 String 的列表也就可以隐式转换为 Object 的列表，这个想法是错误的。C#之所以不允许将 IList<string>转换为 IList<object>是出于类型安全的原因。如果允许转换，则考虑以下代码。

```
IList<string> words = new List<string>("this", "is", "a", "string","array");
IList<object> objects = numbers;                          //假如可以转换成功
objects[0] = 50;
objects[1] = 123.546;
objects[2] = new SaleRecord();
string s = words[2];                                      //出错
```

假如可以将 IList<string>转换为 IList<object>，由于 IList 是引用类型，上述代码中的 objects 和 words 其实是同一个对象。通过 objects 可以向列表中添加任意类型的对象，然后通过 words 试图得到一个 string 类型时就会出错。

虽然 IList<string>不可以转换为 IList<object>，但是 IEnumerable<string>却可以转换为 IEnumerable<object>，代码如下：

```
IEnumerable<string> words = new List<string> { "this", "is", "a", "string",
"array" };
IEnumerable<object> objects = words;
```

C#允许从 IEnumerable<string>到 IEnumerable<object>的转换，是由于 IEnumerable<T>接口没有修改数据的方法，不会出现前述的类型安全问题。这种从派生类泛型接口向基类泛型接口的转换称为协变。C#是如何知道什么情况下允许协变（如 IEnumerable<T>接口）哪些情况下不允许（如 IList<T>）呢？这是通过泛型定义实现的。查看这两个泛型接口的

定义，可以得到以下代码。

```
public interface IEnumerable<out T> : IEnumerable
{ ... }
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{ ... }
```

由上述代码可以看出，在 IEnumerable 泛型接口的类型参数 T 前面有一个 out 关键字，而 IList 泛型接口则没有这个 out 关键字。泛型接口类型参数 T 前面的 out 表示 T 只出现在接口的输入位置（如方法返回值），而不会出现在输入位置（如方法输入参数）。像这种在类型参数前有 out 关键字的泛型接口允许协变。

与协变相对的概念称为逆变，即将一个基类型的泛型接口转换为派生类型的泛型接口，代码如下：

```
IComparer<object> objectComparer=getObjectComparer();
IComparer<string> stringComparer=getStringComparer();
stringComparer = objectComparer;                        //逆变
```

上述代码的第三行，将一个 IComparer<object>类型隐式转换为 IComparer<string>类型。这个转换与通常所理解的类型转换有点矛盾，通常允许从派生类向基类的隐式转换，基类却不能隐式转换为派生类型。对于 IComparer<T>接口来说，由基类接口向派生类接口转换是有意义的，如代码中一个可以比较 Object 类型的比较器，自然可以成为比较 string 类型的比较器。查看 IComparer 泛型接口的定义，可得到如下代码。

```
public interface IComparer<in T>
{ ... }
```

通过上述代码可看出，在 IComparer 泛型接口的类型参数 T 前面有一个 in 关键字，表示在此泛型接口中类型参数 T 只允许出现在输入位置（如方法的输入参数），而不允许出现在输出位置（如方法返回值）。像这种在类型参数前有 in 关键字的泛型接口允许逆变。

提示：用于指示协变和逆变的 in、out 关键字只能用于修饰泛型接口和委托的类型参数，不能用于类、结构、方法等。所修饰的类型 T 必须是引用类型，不能是值类型。

7.4　小　　结

伴随着 Visual Studio 2010 的发布，.NET Framework、ASP.NET、C#都升级到了 4.0 版本，增加了一些新功能。本章介绍了 Visual Studio 2010 IDE 新增功能、ASP.NET 的新特性和 C# 4.0 的新语法。

第 8 章　LINQ 与实体框架 Entity Framework

数据访问一直是大多数应用程序的一个非常重要的工作。为了使数据访问的编码变得简单、高效，软件开发领域不断推出新的数据访问框架和产品。用于.NET 领域的数据访问框架有 Enterprise Library、ActiveRecord、NHibernate、SubSonic 等，ASP.NET 中也包含许多数据访问相关的控件，如各种数据源控件和数据绑定控件。微软公司推出的最新数据访问技术为语言集成查询（Language-Integrated Query 简称 LINQ）和实体框架（Entity Framework 简称 EF）。

8.1　C#对 LINQ 的支持

LINQ 提供了一种跨各种数据源和数据格式使用数据的一致模型。在 LINQ 查询中，始终使用对象而非针对某种具体数据源的操作命令，可以使用相同的基本编码模式来查询和转换 XML 文档、SQL 数据库、ADO.NET 数据集、.NET 集合中的数据及对其有 LINQ 提供程序可用的任何其他格式的数据。LINQ 是一种全新的数据查询方式，C#语言为全面支持 LINQ 增加了一些新的语法和功能，本节将对此进行介绍。

8.1.1　对象初始化器

在编程过程中经常用到的一个功能就是在创建对象时为对象的某些属性赋值。C#的对象初始化器（Object Initializer）可以很好地实现这一功能。其语法如下：

```
new 类名(){属性 1=值 1, 属性 2=值 2, … , 属性 N=值 N};
```

下面通过例子来具体讲解如何在创建类的实例时为其属性赋值。有一个 Student 类，定义如下：

```
class Student
{
   public string name { get; set; }
   public int age { get; set; }
}
```

在创建 Student 类的实例时，可以给 name 属性、age 属性或者二者同时赋值，代码如下：

```
//创建一个 Student 类的实例，并设置其 name 属性
Student s1 = new Student(){ name="John"};
```

上面一行代码的功能相当于以下代码：

```
Student s1 = new Student();
s1.name = "Jonh";
```

也可以在创建类的实例时同时为多个属性赋值，代码如下：

```
Student s2 = new Student {age=20,name="Mary" };
```

8.1.2　隐式类型

使用 LINQ 进行查询时，很多时候编程人员不容易判断某个查询返回的类型，或者类型名称太长不方便写出来，C#中引入了隐式类型这个概念以解决此问题。在 C#中可以使用 var 关键字来声明隐式类型的局部变量，其语法如下：

```
var 变量名=初始值;
```

通过使用 var 关键字，就可以很方便地表示 LINQ 的查询结果。如下代码所示，等号右面的所有代码为一个 LINQ 查询语句，这个查询语句返回一个不能知道名称的类型（称为匿名类型），这种情况下必须使用 var 关键字才能保存这个返回值。

```
var query=  from p in products
            where p.price > 20 && p.price < 100
            select p ;
```

用 var 关键字声明的变量可以被初始化成任何类型的值，如下面的代码所示。

```
//用 int 值初始化 var 变量
var var1 = 21;
//用 string 值初始化 var 变量
var var2 = "some text";
//用 DateTime 值初始化 var 变量
var var3 = DateTime.Now;
//用数组初始化 var 变量
var var4=new object[10];
//把 var 变量初始化为列表
var var4 = new List<string>();
```

在使用 var 关键字定义变量时，需要注意以下两点。

（1）使用 var 关键字定义的变量必须被初始化。如下面的代码是错误的。

```
static void Main(string[] args)
{
    var num;  //var 变量未初始化
    num = 10;
}
```

（2）使用 var 关键字只能声明局部变量（包括在 for、foreach、using 语句中使用的变量），而不能用于声明其他变量，如下面的例子所示。

```
class VarSample
{
```

```
    //错误，var 不能声明类中的字段
    var name = "名字";
    //错误，var 不能声明方法参数
    private void method1(var i)
    {
        Console.WriteLine(i);
    }
    private void method2()
    {
        //正确，var 可用于 for 循环中声明计数变量
        for (var i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
        }
        int[] nums = new int[] { 1, 2, 3, 4 };
        //正确，var 可用于 foreach 语句中的循环变量
        foreach (var num in nums)
        {
            Console.WriteLine(num);
        }
        //正确，var 可用于 using 语句中的变量声明
        using (var file =new System.IO.StreamWriter("e:\\temp.text"))
        {
            file.WriteLine("some text");
        }
    }
}
```

（3）var 变量和 object 类型变量完全不同。使用 object 类型声明的变量是弱类型，可以被赋予任何类型的值，而使用 var 关键字声明的变量与普通变量一样，仍然是强类型变量。var 变量被初始化时，其类型即被确定。

（4）var 变量与 dynamic 变量完全不同。var 变量是一种静态类型的变量，而 dynamic 是一种动态类型变量。静态类型可以执行类型检查、给出智能提示，而动态类型则不可以。在 Visual Studio 开发环境中，智能识别系统能够识别 var 变量的类型，从而给出成员提示，如图 8.1 所示。

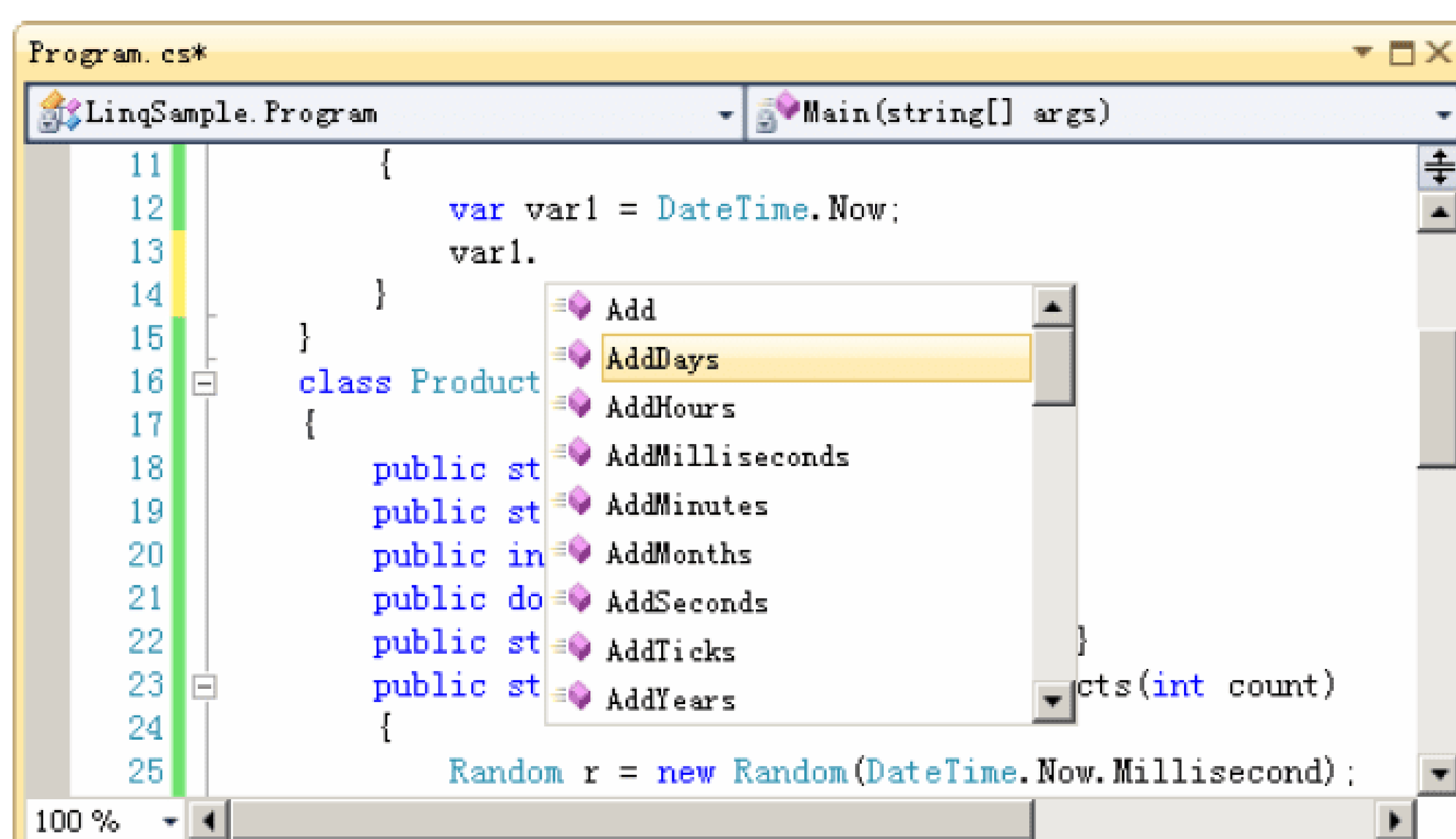


图 8.1　var 变量为强类型变量

从图 8.1 中可以看出，var1 是一个用 var 声明的变量，并且被初始化成 DateTime 类型，在此后使用 var1 变量时，Visual Studio 能够自动识别出 var1 为类型并列出 DateTime 的成员。

由于 var 变量是强类型变量，所以在 var 变量被初始化后，不能再给此变量赋其他类型不兼容的值，如下面的代码所示。

```
//var1 被初始化为 DateTime 类型
var var1 = DateTime.Now;
//下面语句是错误的，试图用 string 值赋给 DateTime 变量
var1 = "尝试赋 String 类型";
//var2 被初始化为 int 类型
var var2 = 10;
//以下语句合法，并且 var3 也是 int 类型
var var3 = var2;
//错误，试图把 DateTime 值赋给 int 类型变量
var3 = var1;
```

提示： 用 var 关键字声明的隐式类型变量是强类型变量，在声明时必须赋值，而且其类型一旦确定不能更改。

8.1.3　匿名类型

C#的匿名类型是指没有名字的类型。匿名类型是一个类，直接继承于 object 类。匿名类型没有单独明确的类的定义，而是在创建匿名类的实例时隐式创建的类型。匿名类型的成员是一组可以读写的属性。

下面的代码创建了一个匿名类型和此类型的一个实例。

```
var s1 = new { id="112",name="John", age=30};
```

上述代码创建了一个具有 3 个读写属性的匿名类，并生成此类的一个实例。这行代码在功能上相当于以下代码。

```
//定义一个类
class AnonymousClass1
{
    public string id { get; set; }
    public string name { get; set; }
    public int age { get; set; }
}
//创建此类的实例
AnonymousClass1 s1 = new AnonymousClass1() { id = "112", name = "John", age
= 30 };
```

在同一个程序中，如果两段创建匿名类型对象的代码具有相同的参数（包括参数类型、名称和顺序），那么这两段代码所创建的匿名对象就属于同一个匿名类型，否则二者就是不同的匿名类型。例 8-1 演示匿名类型的使用。

【例 8-1】 使用匿名类型。

```
static void Main(string[] args)
{
    Type type1,type2,type3;                          //3 个 Type 变量
    var s1 = new { id="101",name="John",age=20}; //创建一个匿名类型对象
    type1 = s1.GetType();                           //得到并输出对象的类型名称
    Console.WriteLine("type1:\t"+type1.Name);
```

```
    //可以像访问普通类型一样访问匿名类型的属性
    string id=s1.id;
    string name=s1.name;
    int age=s1.age;
    Console.WriteLine(string.Format("id:{0};name:{1};age:{2}",id,name,ag
    e));
    var s2 = new { id = "102", name = "Mike", age = 25};
                                              //创建一个匿名类型对象
    //s1 和 s2 具有相同的成员，因此属于相同的匿名类型
    type2 = s2.GetType();
    Console.WriteLine("type2:\t"+type2.Name);
    if (type1 == type2)
    {
        Console.WriteLine("type1 和 type2 相同");

        s2 = s1;                              //相同匿名类型的对象之间可以赋值
    }
    var s3 = new { id = "101",age = 20,name = "John"};
                                      //创建另外一个匿名类型对象
    //由于参数不同（顺序不同），s3 和 s1 不属于同一匿名类型
    type3 = s3.GetType();
    Console.WriteLine("type3:\t"+type3.Name);
    if (type1 != type3)
    {
        Console.WriteLine("s1 和 s3 不同");
    }
    Console.ReadLine();
}
```

上述代码的输出结果如下：

```
type1:  <>f__AnonymousType0`3
id:101;name:John;age:20
type2:  <>f__AnonymousType0`3
type1 和 type2 相同
type3:  <>f__AnonymousType1`3
s1 和 s3 不同
```

8.1.4 扩展方法

C#中的扩展方法允许向现有类中添加方法而不用修改现有类的代码。甚至当没有某个类的源代码，也可以通过扩展方法向类中添加方法。在 C#语言规范中，微软公司建议应该慎重使用扩展方法，只有在用普通方法无法完成功能时，才应考虑使用扩展方法。

扩展函数只能声明在静态类（static class）中，扩展函数与普通静态函数的区别在于其方法的第一个参数前面有一个 this 关键字，这个 this 关键字是扩展函数的标志。this 关键字后面的类型为要扩展的类型。声明扩展方法的语法如下：

```
static class 封装类名
{
   [访问修改符] 返回类型 扩展方法名(this 扩展类名 变量名 0
   [,类型 1 变量 1, 类型 2 变量 2, … ,类型 n 变量 n])
   {
      方法体
   }
}
```

其中封装类名是扩展方法的容器，可以是任意合法标识符。扩展类名是要被扩展的类名，即要在其中添加方法的类。this 变量后面的变量为方法参数列表。在定义了扩展方法以后，扩展类名所表示的类型就增加所定义的扩展方法。下面通过一个例子来看扩展方法的使用。

【例 8-2】 扩展方法。

假设要为.NET Framework 中的 string 类型添加一个 isInt 方法，判断某个字符串是否是个整数，以及是否为正整数。

（1）创建一个控制台应用程序。

（2）在项目中添加一个类，重命名为 StringExtension，在 class 关键字前面加上 static 关键字，把类声明成静态类。代码如下：

```
static class StringExtension
{}
```

（3）在类中添加一个 string 类的扩展 isInt，用来判断一个字符串是否整数。代码如下：

```
/// <summary>
/// 判断字符串是否一个整数或正整数
/// </summary>
/// <param name="text">要判断的字符串</param>
/// <param name="positive">是否要求正数</param>
public static bool isInt(this string text, bool positive=false)
{
    int n;
    if (int.TryParse(text, out n))              //如果能把字符串解析为int类型
    {
        if (positive)                            //如果要求正数
        {
            return n > 0;                        //判断解析得到的整数是否大于 0
        }
        return true;
    }
    return false;
}
```

在 isInt 方法的定义中，需要注意以下两点。

- 第一个参数前面有个 this 关键字，表示这是一个扩展方法。
- this 关键字后面是 string 类型，表示这个扩展方法是对 string 类的扩展。

到此为止，已经在 string 类中添加了一个重载的 isInt 方法，可以在代码中使用了。而且这个 isInt 方法可以被 Visual Studio 开发环境所识别，如图 8.2 所示。

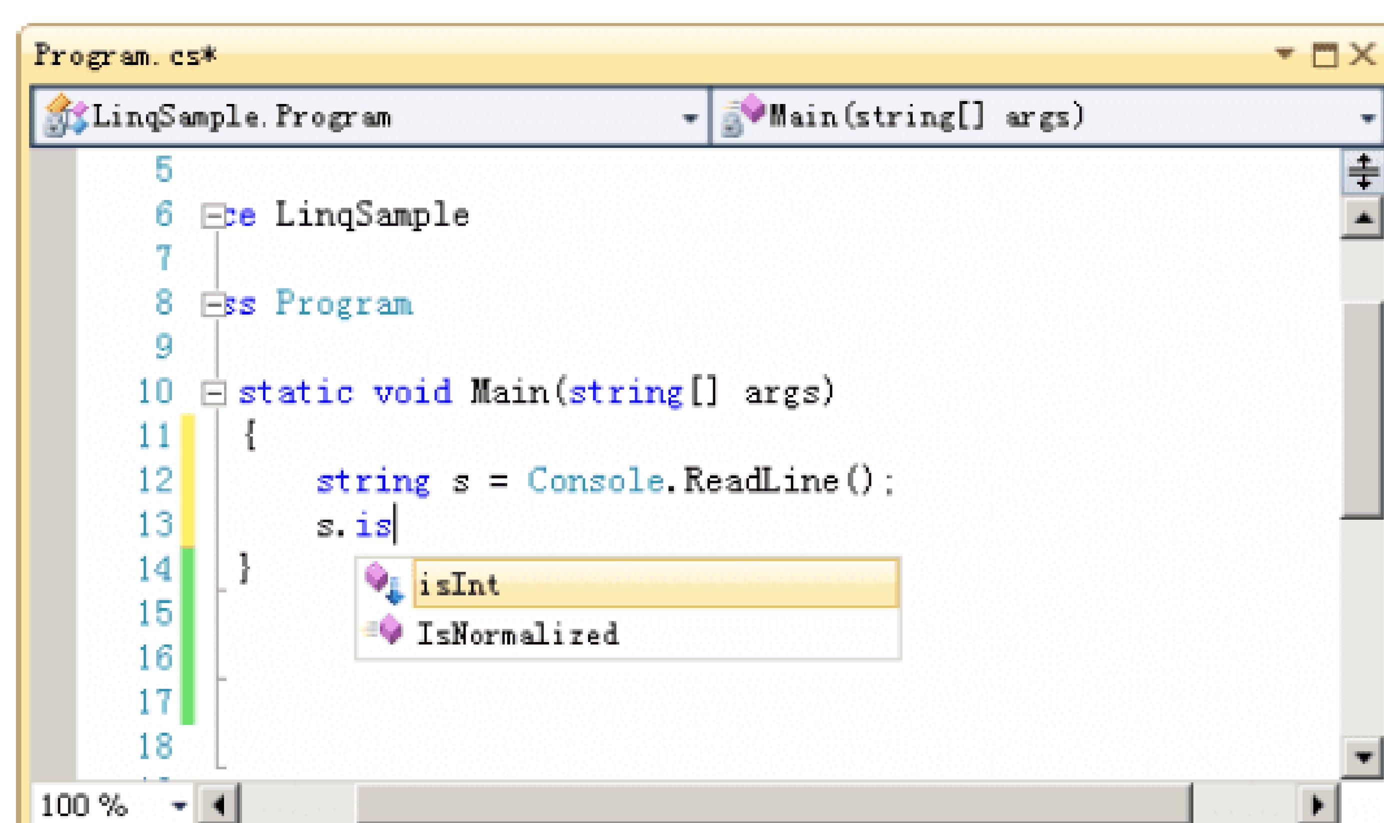


图 8.2　Visual Studio 自动识别扩展方法

（4）在 Main()方法中，输入以下代码，测试 isInt 扩展方法。

```
static void Main(string[] args)
{
    string text = "-9123";
    bool b = text.isInt();
    Console.WriteLine(text + (b ? "" : "不") + "是整数");
    b = text.isInt(true);
    Console.WriteLine(text + (b ? "" : "不") + "是正整数");
    //由于引号中的内容是 string 类型，所以以下代码也合法
    b = "100".isInt();
}
```

（5）运行此程序，输出结果如下：

```
-9123 是整数
-9123 不是正整数
```

8.1.5　Lambda 表达式

Lambda 表达式是对匿名方法的一种改进，具有更加简洁的语法和更易理解的形式。Lambda 表达式可以包含表达式和语句，并且可用于创建委托或表达式目录树类型。所有 Lambda 表达式都使用 Lambda 运算符=>，该运算符读为“goes to”。该 Lambda 运算符的左边是输入参数，右边包含表达式或语句块。声明 Lambda 表示式的语法如下：

```
(参数列表)=>{方法体}
```

其中的参数列表与普通方法的参数列表相同。Lambda 表达式是升级版的匿名方法，其语法形式与匿名方法以及普通方法的语法都有很大的相似之处，如下面的语句所示。

```
返回类型 方法名 (参数列表) { 方法体 }                  //普通方法
delegate (参数列表) { 方法体 }                         //匿名方法
(参数列表) => { 方法体 }                               //Lambda 表达式
```

上述语法描述中，第 1 行是普通的方法声明，第 2 行是匿名方法，第 3 行是 Lambda 表达式。通过三者的对比看出，Lambda 表达式在语法形式上相当于把普通方法声明中的返回类型省略，并且在参数列表和方法体之间加了一个 Lambda 操作符=>。Lambda 表达式的返回类型由语句块中的 return 语句所决定。下面的代码是几个 Lambda 表达式的例子。

```
//求两个整数最大值
(int x, int y) => { return x>y?x:y; }
//决断一个字符串是否数字
(string text) =>
{
   double d;
   return double.TryParse(text, out d);
}
//输入 Hello, world
() => { Console.WriteLine("Hello, World!"); }
```

注意：如果 Lambda 表达式没有参数，则参数列表为空，但必须要有圆括号。

如果 Lambda 表达式只能使用上述语法形式，那么 Lambda 表达式和匿名方法也就没有多大区别。正是由于 Lambda 表达式允许在特定情况下对某些元素省略，使 Lambda 表达式更加灵活易读。Lambda 表达式具有以下省略语法。

（1）如果 Lambda 表达式的参数类型可以通过上下文推断时，参数列表中的类型名称可以被省略。下面的代码是一个省略了参数类型的 Lambda 表达式。

```
(x,y)=>{return x>y?x:y;}
```

（2）如果 Lambda 表达式只有一个参数且参数类型被省略，则参数列表外面的圆括号也可以省略，如下代码所示。

```
x=>{return x++;}
```

（3）如果 Lambda 语句的方法体只有一条 return 语句，且 return 语句有返回类型，则 return 关键字、分号、大括号都可以省略，此时的 Lambda 表达式的方法体只剩下一个表达式。下面是两个例子。

```
(x,y)=>x>y?x:y
//上述代码相当于 (x,y)=>{return x>y?x:y;}
x=>x++
//上面一行代码相当于 x=>{return x++;}
```

Lambda 表达式用的最多的地方就是集合操作，如元素查找。下面给出一个集合操作的例子，说明 Lambda 表达式的应用。

【例 8-3】 Lambda 表达式应用。

本例创建一个学生成绩的列表，并利用 Lambda 表达式作为查询条件，对学生成绩列表进行查找、删除等操作。

```
class Program
{
    static void Main(string[] args)
    {
        List<int> scores = new List<int>();            //学生成绩列表
        //随机填充学生成绩
        Random rand = new Random();
        for (int i = 0; i < 50; i++)
            scores.Add(rand.Next(20, 100));
        Console.WriteLine("原始数据");
        printScores(scores);
        //查找 90 分以上的人
        int match = scores.Count(n => n>=90);
        Console.WriteLine(match+"人成绩高于 90 分");
        //删除不及格的学生成绩
        scores.RemoveAll(n => n<60);
        Console.WriteLine("删除不及格成绩后的数据");
        printScores(scores);
        //按照从高到低的顺序排列学生成绩
        scores.Sort((x, y) => y - x);
        Console.WriteLine("排序后的数据");
        printScores(scores);
    }
    //输出学生成绩，每行 15 个
    static void printScores(List<int> scores)
```

```
    {
        for (int i = 0; i < scores.Count; i++)
        {
            if (i>0 && i % 15 == 0)
                Console.WriteLine();
            Console.Write(string.Format("{0,-4}",scores[i]));
        }
        Console.WriteLine();
    }
}
```

上述代码运行结果如下：

```
原始数据
80  53  91  44  72  42  94  36  29  33  89  43  22  22  47
91  92  37  30  55  58  79  23  64  68  46  50  44  43  37
40  24  49  47  40  68  40  48  53  51  96  49  33  69  58
60  77  47  26  36
5 人成绩高于 90 分
删除不及格成绩后的数据
80  91  72  94  89  91  92  79  64  68  68  96  69  60  77
排序后的数据
96  94  92  91  91  89  80  79  77  72  69  68  68  64  60
```

8.1.6　表达式树

表达式树（Expression Tree）以数据形式表示语言级别代码，数据存储在树形结构中。表达式目录树中的每个节点都表示一个表达式，例如一个方法调用或诸如 x<y 的二元运算。System.Linq.Expressions.Expression 类表示一个表达式，Expression 类是一个抽象类，可提供用于对表达式目录树进行建模的类层次结构的根目录。从 Expression 派生的 System.Linq.Expressions 命名空间中的类（例如 MemberExpression 和 ParameterExpression 等）用于表示表达式目录树中的节点。Expression 类包含一系列 static 工厂方法，可创建各种类型的表达式目录树节点。Expression 主要有以下派生类。

- BinaryExpression：表示包含二元运算符的表达式。
- ConditionalExpression：表示包含条件运算符的表达式。
- ConstantExpression：表示具有常量值的表达式。
- Expression<TDelegate>：以表达式目录树的形式将强类型 Lambda 表达式表示为数据结构。
- InvocationExpression：表示将委托或 Lambda 表达式应用于参数表达式列表的表达式。
- LambdaExpression：描述一个 Lambda 表达式。
- ListInitExpression：表示包含集合初始值设定项的构造函数调用。
- MemberExpression：表示访问字段或属性。
- MemberInitExpression：表示调用构造函数并初始化新对象的一个或多个成员。
- MethodCallExpression：表示调用一种方法。
- NewArrayExpression：表示创建新数组并可能初始化该新数组的元素。
- NewExpression：表示构造函数调用。

- ParameterExpression：表示命名的参数表达式。
- TypeBinaryExpression：表示表达式和类型之间的操作。
- UnaryExpression：表示包含一元运算符的表达式。

下面通过图形的方式形象地表示表达式树的结构。平面几何中求扇形面积的公式为：

Area = angle / 360 * 3.14 * radius * radius，其中 angle 表示扇形圆心角，radius 表示扇形半径。将这个公式化简可以得到：

Area = angle * (3.14 / 360) * radius * radius= (angle * 0.0087) * (radius * radius)

将上述公式用 Lambda 表达式表示，可得

```
Expression<Func<double, double, double>> areaExpression = (r, a) => (a *
0.087) * (r * r);
```

上述表达式所对应的表达式树图形如图 8.3 所示。

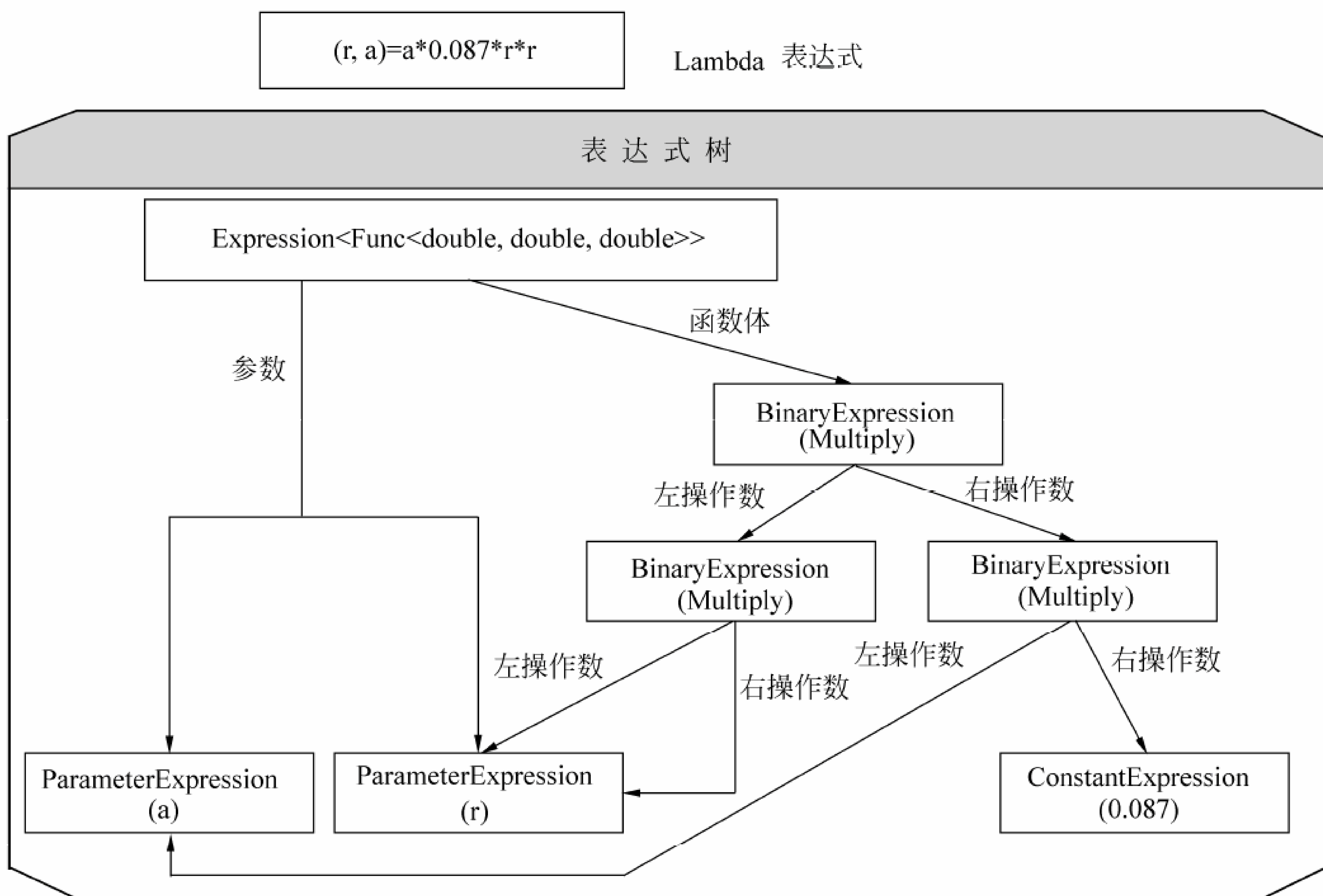


图 8.3　扇形面积表达式树

可以用以下 C#代码构造如图 8.3 所示的表达式树。

```
//构建两个参数表达式（半径 r 和角度 a）
ParameterExpression radiusParam=Expression.Parameter(typeof(double),
"r");
ParameterExpression  angleParam  =  Expression.Parameter(typeof(double),
"a");
//构建常量表达式(3.14/360=0.0087)
ConstantExpression constExp = Expression.Constant(0.0087, typeof(double));
//构建两个乘法表达式 a*0.0087 和 r*r
BinaryExpression mul1 = Expression.Multiply(angleParam, constExp);
BinaryExpression mul2 = Expression.Multiply(radiusParam, radiusParam);
//将上面两个乘法表达式相乘，得到面积计算表达式
```

```
BinaryExpression result = Expression.Multiply(mul1, mul2);
//将面积表达式封装为 Lambda 表达式
Expression<Func<double, double, double>> areaExpression =
    Expression.Lambda<Func<double, double, double>>
    (result, new ParameterExpression[] { radiusParam,angleParam });
```

8.2　LINQ 基本操作

LINQ 提供了一种跨各种数据源和数据格式使用数据的一致模型。为了实现这种一致性，LINQ 充分利用 C#中的扩展方法、Lambda 表达式、对象初始化器、匿名类型等特性。LINQ 为 IEnumerable<T>泛型接口添加了若干扩展方法，.NET 中所有泛型集合类都实现了 IEnumerable 接口，通过这些扩展方法可以实现对泛型集合的查询功能。

LINQ 的查询有两种语法，一种是方法语法，即使用传统的 C#类和方法的语法进行查询，另外一种语法是查询语法，这是为 LINQ 而引入的新的 C#语法。这两类语法形式不同，但是功能相同。实际上 C#在遇到 LINQ 的查询语法时，会翻译成相应的方法语法再执行。

8.2.1　创建查询数据源

LINQ 用于从各种数据源中查询数据。为了演示 LINQ 的语法和功能，需要创建一部分数据，以这些数据为基础进行查询。本节将使用一组随机生成的商品销售数据为数据源演示 LINQ 的使用。与此相关的有 3 个类：商品类别 Category、商品信息 Product 和销售记录 ProductSale。为了随机生成数据，3 个类中分别包含一个随机填充数据的方法。Category 类的代码如下：

```
public class Category
{
    public string id { get; set; }                          //类别 ID
    public string name { get; set; }                        //类别名称
    /// <summary>
    /// 随机生成商品类别
    /// </summary>
    /// <param name="count">要生成的商品类别数量</param>
    /// <returns>生成的商品类别列表</returns>
    public static List<Category> randomCategories(int count)
    {
        Random r=new Random(DateTime.Now.Millisecond);
        List<Category> categories = new List<Category>();
        //循环生成类别数据，不保证类别 ID 唯一
        for (int i = 0; i < count; i++)
        {
            Category category = new Category();
            category.id = "C" + r.Next(1000);
            category.name = "Category " + i.ToString();
            categories.Add(category);
        }
        return categories;
    }
}
```

Product 类的代码如下：

```
public class Product
{
    public string id { get; set; }                                    //商品 ID
    public string name { get; set; }                                  //名称
    public int storage { get; set; }                                  //库存
    public double price { get; set; }                                 //价格
    public Category category { get; set; }                            //类别
    /// <summary>
    /// 随机生成商品列表
    /// </summary>
    /// <param name="count">要生成的商品数量</param>
    /// <returns>所生成的商品列表</returns>
    public static List<Product> randomProducts(int count)
    {
        Random r = new Random(DateTime.Now.Millisecond);
        List<Product> products = new List<Product>();
        //随机生成类别，设类别数量为商品数量的 1/10，但至少有 3 种，最多有 20 种
        int categoryCount = Math.Min(20, Math.Max(3, count / 10));
        Category[] categories = Category.randomCategories(categoryCount).
         ToArray();
        for (int i = 0; i < count; i++)
        {
            Product p = new Product();
            p.id = "P" + r.Next(10000).ToString("0000");
            p.name = "Product " + i.ToString();
            p.storage = r.Next(10, 10000);
            p.price = 1 + r.NextDouble() * 300;
            p.category = categories[r.Next(categoryCount)];
            products.Add(p);
        }
        return products;
    }
}
```

ProductSale 类的代码如下：

```
public class ProductSale
{
    public Product product { get; set; }                   //所销售商品
    public DateTime date { get; set; }                     //销售日期
    public int quantity { get; set; }                      //销售数量
    /// <summary>
    /// 随机生成一组销售数据
    /// </summary>
    /// <param name="count">要生成的销售记录的数量</param>
    /// <returns>所生成的销售数据列表</returns>
    public static List<ProductSale> randomSales(int count)
    {
        //随机生成商品列表，设商品种类为销售记录的 1/5
        int productCount = count / 5 +1;
        Product[] products = Product.randomProducts(productCount).
         ToArray();
        Random r = new Random(DateTime.Now.Millisecond);
        List<ProductSale> sales = new List<ProductSale>();
        for (int i = 0; i < count; i++)
        {
```

```
                ProductSale sale = new ProductSale();
                sale.product = products[r.Next(productCount)];
                sale.quantity = r.Next(100);
                sale.date = DateTime.Now.AddDays(r.Next(-30, 30));
                sales.Add(sale);
            }
            return sales;
        }
}
```

8.2.2　投影

在数据查询中投影是指从一个数据源中查询某些字段或属性，例如 SQL 语句中的“select 列名 1, 列名 2, …, 列名 N from 表名”就是一个投影操作。在 LINQ 中，IEnumerable<T>类的 Select 扩展方法可以实现投影操作，方法声明如下：

```
//摘要:将序列中的每个元素投影到新表中
//参数:
//source:一个值序列，要对该序列调用转换函数
//selector:应用于每个元素的转换函数
//类型参数:
//TSource: source 中的元素的类型
//TResult: selector 返回的值的类型
//返回结果: 一个 System.Collections.Generic.IEnumerable<T>,其元素为对 source
的每个元素调用转换函数的结果
//异常:System.ArgumentNullException: source 或 selector 为 null
public   static   IEnumerable<TResult>   Select<TSource,   TResult>(this
IEnumerable<TSource> source, Func<TSource, TResult> selector);
```

Select 方法用于从集合中查询数据，所查询到的数据并不一定与集合中原有数据类型相同，而是可以基于原有数据创建任意的数据类型。例如，可以从一个学生信息集合中查询所有的学号。Select 方法参数中的 Func<TSource, TResult>实现了这个类型转换功能，通常为这个参数传递一个 Lambda 表达式。

【例 8-4】 Select 查询。

本例演示如何使用 IEumerable<T>类的 Select 扩展方法查询商品信息。

```
public void selectDemo()
{
    List<Product> products = Product.randomProducts(5);
    var query = products.Select(p => p);                    //查询商品本身
    foreach (var item in query)
    {

    Console.WriteLine("Id={0},Name={1},Price={2},storage={3},category=
    {4}",
            item.id, item.name, item.price, item.storage, item.category.
            name);
    }
    //查询商品的一部分信息，这些信息构成一个匿名类型
    var query2 = products.Select(p => new { Id = p.id, Name = p.name, Category
    = p.category.name });
    foreach (var item in query2)
    {
```

```
        Console.Write("Id={0},Name={1},Category={2}", item.Id, item.Name,
       item.Category);
    }
}
```

上述代码运行结果如下：

```
Id=P3849,Name=Product
0,Price=123.874712721852,storage=7362,category=Category 0
Id=P7290,Name=Product 1,Price=263.713322072622,storage=4351,category=
Category 2
Id=P0212,Name=Product 2,Price=10.2506326778096,storage=3186,category=
Category 0
Id=P0479,Name=Product 3,Price=21.8219276838107,storage=1460,category=
Category 2
Id=P2850,Name=Product 4,Price=283.019849439161,storage=8551,category=
Category 2
Id=P3849,Name=Product 0,Category=Category 0
Id=P7290,Name=Product 1,Category=Category 2
Id=P0212,Name=Product 2,Category=Category 0
Id=P0479,Name=Product 3,Category=Category 2
Id=P2850,Name=Product 4,Category=Category 2
```

如前所述，LINQ 有两种语法：方法语法和查询语法。LINQ 的查询语法如下：

```
from 变量名 in 数据源
select 表达式
```

例 8-4 所使用的语法为扩展方法语法，所对应的等价查询语法如下：

```
public void selectDemo()
{
    List<Product> products = Product.randomProducts(5); //随机生成商品列表
    var query = from p in products
                select p;
    foreach (var item in query)
    {

    Console.WriteLine("Id={0},Name={1},Price={2},storage={3},category={4
    }",
            item.id, item.name, item.price, item.storage, item.category.
            name);
    }
    //查询商品的一部分信息，这些信息构成一个匿名类型
    var query2 = from p in products
                 select new { Id = p.id, Name = p.name, Category = p.category.
                 name };
    foreach (var item in query2)
    {
        Console.WriteLine("Id={0},Name={1},Category={2}", item.Id, item.
        Name, item.Category);
    }
}
```

8.2.3　选择

数据查询中的选择是指从数据源中查找符合条件的数据，过滤掉不符合条件的数据。

SQL 语言中 Select 语句后面的 Where 条件所起的作用就是选择。在 LINQ 中 IEnumerable<T>类的 Where 扩展方法可以实现数据过滤功能，Where 方法声明如下：

```
//摘要：基于谓词筛选值序列
//参数：
//source:要筛选的 System.Collections.Generic.IEnumerable<T>
//predicate:用于测试每个元素是否满足条件的函数
//类型参数：
//TSource: source 中的元素的类型
//返回结果：
//一个 System.Collections.Generic.IEnumerable<T>，包含输入序列中满足条件的元素
//异常： System.ArgumentNullException:  source 或 predicate 为 null
public static IEnumerable<TSource> Where<TSource>(this IEnumerable
<TSource> source, Func<TSource, bool> predicate);
```

与 Where 扩展方法相对应的查询语法为：

```
from 变量名 in 数据源
where 条件
select 表达式
```

【例 8-5】 Where 过滤数据。

本例演示使用 Where 方法查询价格大于 180 元的商品信息，代码中同时给出了方法语法和查询语法两种形式，其功能是等价的。

```
public void whereDemo()
{
    List<Product> products = Product.randomProducts(10);//随机生成商品列表
    var query = from p in products
                where p.price > 180
                select p;                              //LINQ 查询
    //或者使用查询方法：var query=products.Where(p => p.price > 180);
    foreach (var item in query)
    {
        Console.WriteLine("Id={0},Name={1},Price={2}",
           item.id, item.name, item.price);
    }
    var query2 = from p in products
                 where p.price > 180
                 select new { Id = p.id, Name = p.name, Category = p.cate-
                 gory.name };
    //或者使用查询方法，如下代码所示
    //var query2 = products.Where(p => p.price > 180)
    //    .Select(p => new { Id = p.id, Name = p.name, Category = p.
           category.name });
    foreach (var item in query2)
    {
        Console.WriteLine("Id={0},Name={1},Category={2}", item.Id, item.
        Name, item.Category);
    }
}
```

上述代码运行结果如下：

```
Id=P1569,Name=Product 1,Price=199.694056784126
Id=P1562,Name=Product 2,Price=240.83325019471
Id=P6013,Name=Product 3,Price=290.243594738303
```

```
Id=P1739,Name=Product 9,Price=284.309072900242
Id=P1569,Name=Product 1,Category=Category 1
Id=P1562,Name=Product 2,Category=Category 0
Id=P6013,Name=Product 3,Category=Category 2
Id=P1739,Name=Product 9,Category=Category 1
```

8.2.4　排序

为了实现数据排序，IEnumerable<T>类包含 4 个相关的扩展方法，分别为 OrderBy 方法、OrderByDescending 方法、ThenBy 方法、ThenByDescending 方法。其中 OrderBy 方法的功能是将数据源按照一个表达式进行升序排序，OrderByDescending 方法的作用是按照一个表达式进行降序排序。ThenBy 方法的作用是对一个已经按照某表达式排序的集合再按照另一个表达式进行排序，henByDescending 方法的功能与之类似，只是按照降序排序。OrderBy 方法声明如下代码所示，其他 3 个方法声明与之类似。

```
// 摘要:根据键按升序对序列的元素排序
// 参数:
// source: 一个要排序的值序列
// keySelector:用于从元素中提取键的函数
// 类型参数:
// TSource: source 中的元素的类型
// TKey:keySelector 返回的键的类型
// 返回结果:一个 System.Linq.IOrderedEnumerable<TElement>，其元素按键排序
// 异常: System.ArgumentNullException: source 或 keySelector 为 null
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>
(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
```

与 OrderBy 等 4 个排序方法相对应的查询语法如下：

```
from 变量名 in 数据源
where 条件
orderby 表达式 1 [descending] ,表达式 2 [descending] , … ,表达式 n [descending]
select 表达式
```

【例 8-6】 数据排序。

本例演示按照单关键字和多关键字对商品信息进行排序。

```
public void sortDemo()
{
    List<Product> products = Product.randomProducts(10);
    //把商品按照价格排序（升序）
    var query = from p in products
                orderby p.price
                where p.price>180
                select p;
    foreach (var item in query)
    {
        Console.WriteLine("Id={0},Name={1},Price={2}",
          item.id, item.name, item.price);
    }
    //上述语句也可以采用查询方法写成以下形式
    //var query=products.OrderBy(p => p.price);
    //把商品先按照类别编号（升序）和价格（降序）排序
    var query2 = from p in products
```

```
                    where p.price<100
                    orderby p.category.id,p.price descending
                    select p;
    //上述语句也可以采用查询方法写成以下形式
    //var query2 = products.OrderBy(p => p.category.id).ThenByDescending(p
    => p.price);
    foreach (var item in query2)
    {
        Console.WriteLine("Id={0},Name={1},Price={2}",
           item.id, item.name, item.price);
    }
}
```

8.2.5　数据分页

在查询结果包含大量数据时，出于性能方面的考虑，通常需要将查询结果分页显示。IEnumerable<T>类的 Skip 扩展方法和 Take 扩展方法很方便地能够实现数据分页功能。Take 方法的作用是从序列的开头返回指定数量的连续元素，Skip 方法的作用是跳过序列中指定数量的元素，然后返回剩余的元素，这两个方法配合使用，先跳过一定数量的元素，再取一定数量的元素，就实现了分页功能。Skip 方法和 Take 方法声明如下。

```
//摘要：跳过序列中指定数量的元素，然后返回剩余的元素
//参数:
//source: 要从中返回元素的 System.Collections.Generic.IEnumerable<T>
//count:  返回剩余元素前要跳过的元素数量
//返回结果:
//一个 System.Collections.Generic.IEnumerable<T>，包含输入序列中指定索引后出现
的元素
//异常:  System.ArgumentNullException: source 为 null
public static IEnumerable<TSource> Skip<TSource>(this IEnumerable<TSource>
source, int count);
//摘要：从序列的开头返回指定数量的连续元素
//参数:
//source: 要从其返回元素的序列
//count: 要返回的元素数量
//返回结果:
//一个 System.Collections.Generic.IEnumerable<T>，包含输入序列开头的指定数量的
元素
//异常: System.ArgumentNullException: source 为 null。
public static IEnumerable<TSource> Take<TSource>(this IEnumerable<TSource>
source, int count);
```

【例 8-7】　数据分页。

本例演示 LINQ 的数据分页功能，查找所有商品中单价大于 200 元的商品，把查询结果按价格降序排列，并分页输出。

```
public void skipTake()
{
    List<Product> products = Product.randomProducts(100);
    //查找商品列表中单价高于 200 的商品，按照价格降序排列
    var query = from p in products
                where p.price > 200
                orderby p.price descending
```

```
                select p;
    //设置页面大小为 3，并计算总页数
    int pageSize = 3;
    int pageCount = (int)Math.Ceiling(query.Count() *1.0 / pageSize);
    //循环输出每页数据
    for (int i = 0; i < pageCount; i++)
    {
        Console.WriteLine("======第{0}页数据======",i+1);
        var query2 = query.Skip(i*pageSize).Take(pageSize);
        foreach (var item in query2)
        {
            Console.WriteLine("Id={0},Name={1},Price={2}",
               item.id, item.name, item.price);
        }
    }
}
```

上述代码部分输出结果如下：

```
======第 1 页数据======
Id=P2721,Name=Product 85,Price=298.89590802877
Id=P5429,Name=Product 35,Price=294.308188157765
Id=P3227,Name=Product 56,Price=286.606975707042
======第 2 页数据======
Id=P6665,Name=Product 70,Price=281.919481292795
Id=P2705,Name=Product 72,Price=276.074729218648
Id=P7969,Name=Product 22,Price=275.414076830453
```

8.2.6　数据分组

进行数据查询时很多时候需要对数据进行分组，IEnumerable<T>类的 GroupBy 方法实现了分组功能。GroupBy 方法声明如下。

```
//摘要：根据指定的键选择器函数对序列中的元素进行分组
//参数：
//source:要对其元素进行分组的 System.Collections.Generic.IEnumerable<T>
//keySelector: 用于提取每个元素的键的函数
//类型参数：
//TSource:source 中的元素的类型
//TKey:keySelector 返回的键的类型
//返回结果：
//IEnumerable<IGrouping<TKey, TSource>>，其中每个 System.Linq.IGrouping
<TKey,TElement> 对象都包含一个对象序列和一个键
//异常:System.ArgumentNullException: source 或 keySelector 为 null
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource,
TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
```

【例 8-8】 分组查询。

本例演示分组查询功能，将所有商品按照类别 ID 进行分组，并输出各组的键值（类别 ID）和各组中的商品列表。

```
public void groupDemo()
{
    List<Product> products = Product.randomProducts(6);
    //将商品按照类别 id 分组
```

```
    var query = from p in products
                group p by p.category.id
                    into categoryGroup
                    select new { categoryid = categoryGroup.Key, products
                    = categoryGroup };
    int i = 1;
    //循环输出每组的类别 ID 和组中的商品
    foreach (var group in query)
    {
        Console.WriteLine("========第{0}组(类别 id:{1})========", i++,
        group.categoryid);
        foreach (var item in group.products)
        {
            Console.WriteLine("Id={0},Name={1},Price={2}",
               item.id, item.name, item.price);
        }
    }
}
```

上述代码输出结果如下：

```
========第 1 组(类别 id:C810)========
Id=P2245,Name=Product 0,Price=244.044761169257
Id=P6378,Name=Product 1,Price=241.861566942586
Id=P4383,Name=Product 4,Price=271.048425844893
========第 2 组(类别 id:C224)========
Id=P2773,Name=Product 2,Price=67.5605930921438
========第 3 组(类别 id:C590)========
Id=P8750,Name=Product 3,Price=54.3286917737353
Id=P5447,Name=Product 5,Price=108.005137254952
```

8.2.7　返回单个元素

根据主键的值进行查询时，查询结果最多只有一项，如下面的 SQL 语句至多返回一条记录（StudentId 为主键）：

```
Select * from Student where StudentId='BZXY10110201'
```

对于这种最多只返回一条数据的查询，在 LINQ 中可以使用 IEnumerable<T>的 Single 和 SingleOrDefault 扩展方法。Single 方法用于读取不多不少单条记录，如果查询结果多于或者少于一条记录都会引发异常。SingleOrDefault 返回数据源中的单条记录或者返回空（如果数据源为空），如果数据源包含多条数据，则引发异常。Single 和 SingleOrDefault 方法声明如下。

```
//摘要：返回序列的唯一元素；如果该序列并非恰好包含一个元素，则会引发异常
//参数：
//source：一个 System.Collections.Generic.IEnumerable<T>，用于返回单个元素
//类型参数：
//TSource:  source 中的元素的类型
//返回结果：输入序列的单个元素
//异常：
//System.ArgumentNullException:  source 为 null
//System.InvalidOperationException：输入序列包含多个元素。- 或 -输入序列为空
public static TSource Single<TSource>(this IEnumerable<TSource> source);
//摘要：如果该序列包含多个元素，此方法将引发异常
```

```
//返回结果：返回输入序列的单个元素；如果序列不包含任何元素，则返回 default(TSource)
// 异常：
//System.ArgumentNullException:  source 为 null
//System.InvalidOperationException:  输入序列包含多个元素
public static TSource SingleOrDefault<TSource>(this IEnumerable<TSource>
source);
```

对于返回多条记录的查询，如果想只取一条记录，则可以使用 IEnumerable<T>的 First 和 FirstOrDefault 扩展方法。Fist 方法的作用是返回查询结果的首元素，如果查询结果为空，则引发异常。FirstOrDefault 方法的作用是返回查询结果的首元素，如果查询结果为空，则返回元素类型的默认值（通常为 null）。First 和 FirstOrDefault 方法声明如下：

```
//摘要:返回序列中的第一个元素
//参数:
//source:要返回其第一个元素的 System.Collections.Generic.IEnumerable<T>
//类型参数: TSource:  source 中的元素的类型
//返回结果:  返回指定序列中的第一个元素
//异常:
//System.ArgumentNullException:  source 为 null
//System.InvalidOperationException: 源序列为空
public static TSource First<TSource>(this IEnumerable<TSource> source);
//摘要: 返回序列中的第一个元素；如果序列中不包含任何元素，则返回默认值
//返回结果:
//如果 source 为空，则返回 default(TSource)；否则返回 source 中的第一个元素
//异常:  System.ArgumentNullException:  source 为 null
public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource>
source);
```

【例 8-9】 查询单个元素。

本例演示使用 Single、SingleOrDefault、First、FirstOrDefault 方法查询单个元素。首先生成一个商品列表，然后分别调用以上 4 个方法查询商品列表中价格大于 200 元的单个商品。

```
public void singleFirst()
{
    List<Product> products = Product.randomProducts(6);
    //查询商品列表中单价大于 200 的商品
    var query = from p in products
                where p.price > 200
                select p;
    Product item=null;
    //调用 Single 方法进行查询
    try
    {
        item = query.Single();
        Console.WriteLine("Single 方法查询结果为: ");
        Console.WriteLine("Id={0},Name={1},Price={2}",
                item.id, item.name, item.price);
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine("Single 发生异常。查询结果不是正好包含单条数据。");
        Console.WriteLine("异常详细信息: "+ex.Message);
    }
    //调用 SingleOrDefault 方法进行查询
```

```
    try
    {
        item = query.SingleOrDefault();
        Console.WriteLine("SingleOrDefault 方法查询结果为：");
        if (item == null)
            Console.WriteLine("<NULL>");
        else
            Console.WriteLine("Id={0},Name={1},Price={2}",
                    item.id, item.name, item.price);
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine("SingleOrDefault 发生异常。查询结果包含多于一条数据。
        ");
        Console.WriteLine("异常详细信息：" + ex.Message);
    }
    //调用 First 方法进行查询
    try
    {
        item = query.First();
        Console.WriteLine("First 方法查询结果为：");
        Console.WriteLine("Id={0},Name={1},Price={2}",
                item.id, item.name, item.price);
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine("Single 发生异常。查询结果不包含数据。");
        Console.WriteLine("异常详细信息：" + ex.Message);
    }
    //调用 FirstOrDefault()方法进行查询
    item = query.FirstOrDefault();
    Console.WriteLine("FirstOrDefault 方法查询结果为：");
    if (item == null)
        Console.WriteLine("<NULL>");
    else
        Console.WriteLine("Id={0},Name={1},Price={2}",
                item.id, item.name, item.price);

}
```

8.2.8　延迟执行和立即执行

在 LINQ 表达式的 from…where…select 语句中，并没有执行实际的查询工作，而仅仅是生成了一个查询命令，实际的查询要等到使用查询结果时（例如在 foreach 语句中使用查询结果）才被执行。这种现象叫做 LINQ 的延迟执行。如果想让查询立即执行，可以调用查询的 ToList 方法，将查询结果转换为列表，从而强制查询立即执行。下面通过一个例子说明查询的延迟执行和立即执行。

【例 8-10】 LINQ 延迟执行。

（1）创建一个控制台应用程序。

（2）向项目中添加一个 Person 类，代码如下：

```
class Person
{
```

```
    public string name { get; set; }
    public string sex { get; set; }
    public int age { get; set; }
    public override string ToString()
    {
        Console.WriteLine("Person.ToString():\t 姓名"+name);
        //线程休眠 1 秒钟
        System.Threading.Thread.Sleep(1000);
        return string.Format("姓名：{0}性别：{1}年龄：{2}",name,sex,age);
    }
}
```

（3）在 Main 方法中编写以下代码：

```
static void Main(string[] args)
{
    List<Person> people = new List<Person>();
    people.Add(new Person { name = "张三", sex = "男", age = 23 });
    people.Add(new Person { name = "李四", sex = "男", age = 28 });
    people.Add(new Person { name = "小妹", sex = "女", age = 17 });
    people.Add(new Person { name = "赵芳", sex = "女", age = 22 });
    var temp = from p in people
               where p.age>20
               select p.ToString();
    Console.WriteLine("Main 方法:\t 年龄大于 20 的人：");
    foreach (var s in temp)
    {
        Console.WriteLine("Main 方法:\t"+s);
    }
}
```

（4）运行此程序，可以看到，Person.ToString 方法和 Main 方法的代码是交替执行的。也就是说，在 Main 方法的 foreach 语句中，每当使用 temp 结果集中的一个元素时，这个元素才被计算生成，而不是在声明 temp 结果集的时候一次性生成所有元素。程序输出结果如下：

```
Main 方法:    年龄大于 20 的人：
Person.ToString():  姓名张三
Main 方法:    姓名：张三性别：男年龄：23
Person.ToString():  姓名李四
Main 方法:    姓名：李四性别：男年龄：28
Person.ToString():  姓名赵芳
Main 方法:    姓名：赵芳性别：女年龄：22
```

（5）由于 LINQ 查询的延迟执行，所以在读取 LINQ 查询结果集时，其中的元素总是最新的，即使在定义了 LINQ 查询以后，源数据集合发生了改变。下面通过实际例子来说明这一点。修改 Main 方法的代码如下：

```
static void Main(string[] args)
{
    List<Person> people = new List<Person>();
    people.Add(new Person { name = "张三", sex = "男", age = 23 });
    people.Add(new Person { name = "李四", sex = "男", age = 28 });
    people.Add(new Person { name = "小妹", sex = "女", age = 17 });
    people.Add(new Person { name = "赵芳", sex = "女", age = 22 });
```

```
    //此时有 3 个人年龄大于 20 岁
    var temp = from p in people
             where p.age>20
             select p.ToString();
    Console.WriteLine("Main 方法:\t 年龄大于 20 的人: ");
    foreach (var s in temp)
    {
        Console.WriteLine("Main 方法:\t"+s);
    }
    Console.WriteLine("Main 方法:\t 把所有人的年龄减小 5 岁");
    //把所有人的年龄减小 5 岁
    people.ForEach(p => p.age -= 5);
    //此时有 1 个人年龄大于 20 岁
    foreach (var s in temp)
    {
        Console.WriteLine("Main 方法:\t"+s);
    }
}
```

（6）再次运行程序，此次程序输出以下结果。由输出结果可以看出，虽然程序中两次输出都使用了同一个查询，但是由于在两次输出之间对年龄进行了修改，得到的输出结果并不相同。

```
Main 方法:    年龄大于 20 的人:
Person.ToString():  姓名张三
Main 方法:    姓名: 张三性别: 男年龄: 23
Person.ToString():  姓名李四
Main 方法:    姓名: 李四性别: 男年龄: 28
Person.ToString():  姓名赵芳
Main 方法:    姓名: 赵芳性别: 女年龄: 22
Main 方法:    把所有人的年龄减小 5 岁
Person.ToString():  姓名李四
Main 方法:    姓名: 李四性别: 男年龄: 23
```

（7）LINQ 查询默认情况下是延迟执行的，若想让 LINQ 查询立即执行，可以调用 ToList 方法和 ToArray 方法，如下面代码所示。

```
var temp = (from p in people
         where p.age>20
         select p.ToString()).ToList();
```

8.3　实体框架 Entity Framework

实体框架是 ADO.NET 中的一组支持开发面向数据的软件应用程序技术。实体框架 使开发人员可以采用特定于域的对象和属性（如客户和客户地址）的形式使用数据，而不必自己考虑存储这些数据的基础数据库表和列，大大提升了数据访问代码的通用性、可读性和编程效率。

8.3.1　实体框架基本概念

数据建模通常将数据模型分为 3 个部分：概念模型、逻辑模型和物理模型。概念模型定义要建模的系统中的实体和关系。关系数据库的逻辑模型通过外键约束将实体和关系规范化到表中。物理模型通过指定分区和索引等存储详细信息实现特定数据引擎的功能。

物理模型由数据库管理员进行优化以改善性能，而编写应用程序代码的程序员工作主要限制为通过编写 SQL 查询和调用存储过程来处理逻辑模型。概念模型通常用作捕获和传达应用程序的要求的工具。

实体框架可使开发人员查询概念模型中的实体和关系，同时依赖于实体框架将这些操作转换为特定于数据源的命令。这使应用程序不再对特定数据源具有硬编码的依赖性。概念模型、存储模型以及两个模型之间的映射以外部规范（称为实体数据模型 EDM）表示。可以根据需要对存储模型和映射进行更改，而不需要对概念模型、数据类或应用程序代码进行更改。存储模型是特定于提供程序的，因此可以在各种数据源之间使用一致的概念模型。

EDM 由以下三种模型和具有相应文件扩展名的映射文件进行定义。

- ❑ 概念架构定义语言文件（.csdl）定义概念模型。
- ❑ 存储架构定义语言文件（.ssdl）定义存储模型（又称逻辑模型）。
- ❑ 映射规范语言文件（.msl）定义存储模型与概念模型之间的映射。

8.3.2　创建数据模型

实体框架基于实体模型进行工作。实体框架实现了模型到数据库之间的双向转换，既可以从数据库生成实体模型，也可以从实体模型自动生成数据库。如果要从数据库生成实体模型，则需要按照以下步骤进行操作。

（1）在 Visual Studio 解决方案资源管理器中，在项目名称上右击，从弹出菜单中选择“添加”|“新建项”命令，在弹出的“添加新项”对话框中选择“ADO.NET 实体数据模型”，如图 8.4 所示。

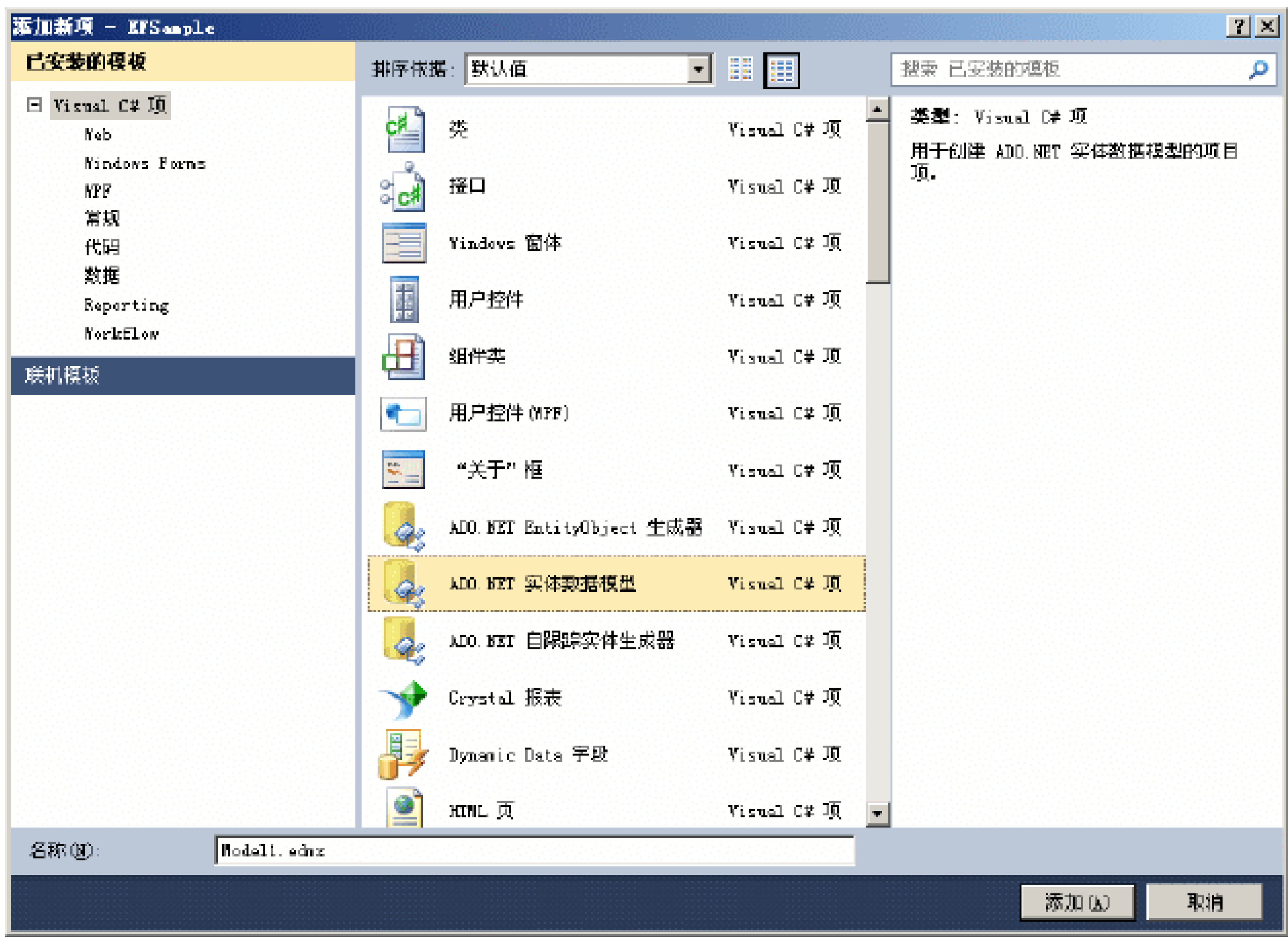


图 8.4　添加 ADO.NET 实体数据模型

（2）在图 8.4 中选择了添加 ADO.NET 实体数据模型以后，将会打开“实体数据模型向导”，在向导第一步选择“从数据库生成”，如图 8.5 所示。

（3）在图 8.5 所示向导中单击“下一步”按钮，进入选择数据库连接对话框，选择 SQL Server 的示例数据库 Northwind。

（4）在向导中单击“下一步”按钮，进入“选择数据库”对象步骤。在此选择 Northwind 表中的 Categories、Products、Orders、Order Details 几个表，并确保“在模型中加入外键列”选项被选中，如图 8.6 所示。



图 8.5　从数据库生成模型

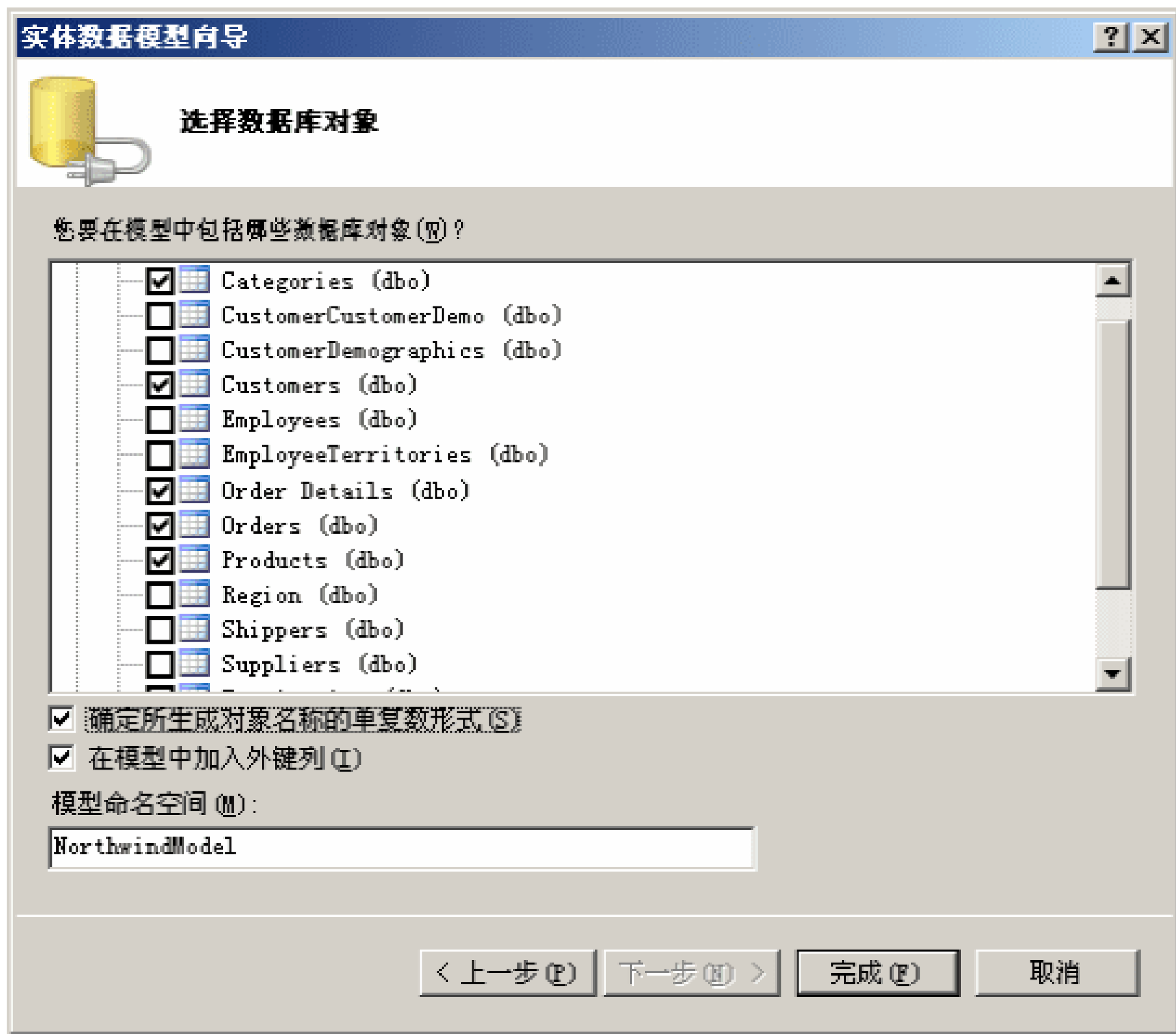


图 8.6　选择数据库对象

（5）单击“完成”按钮，则向导结束后，会生成如图 8.7 所示的实体模型。

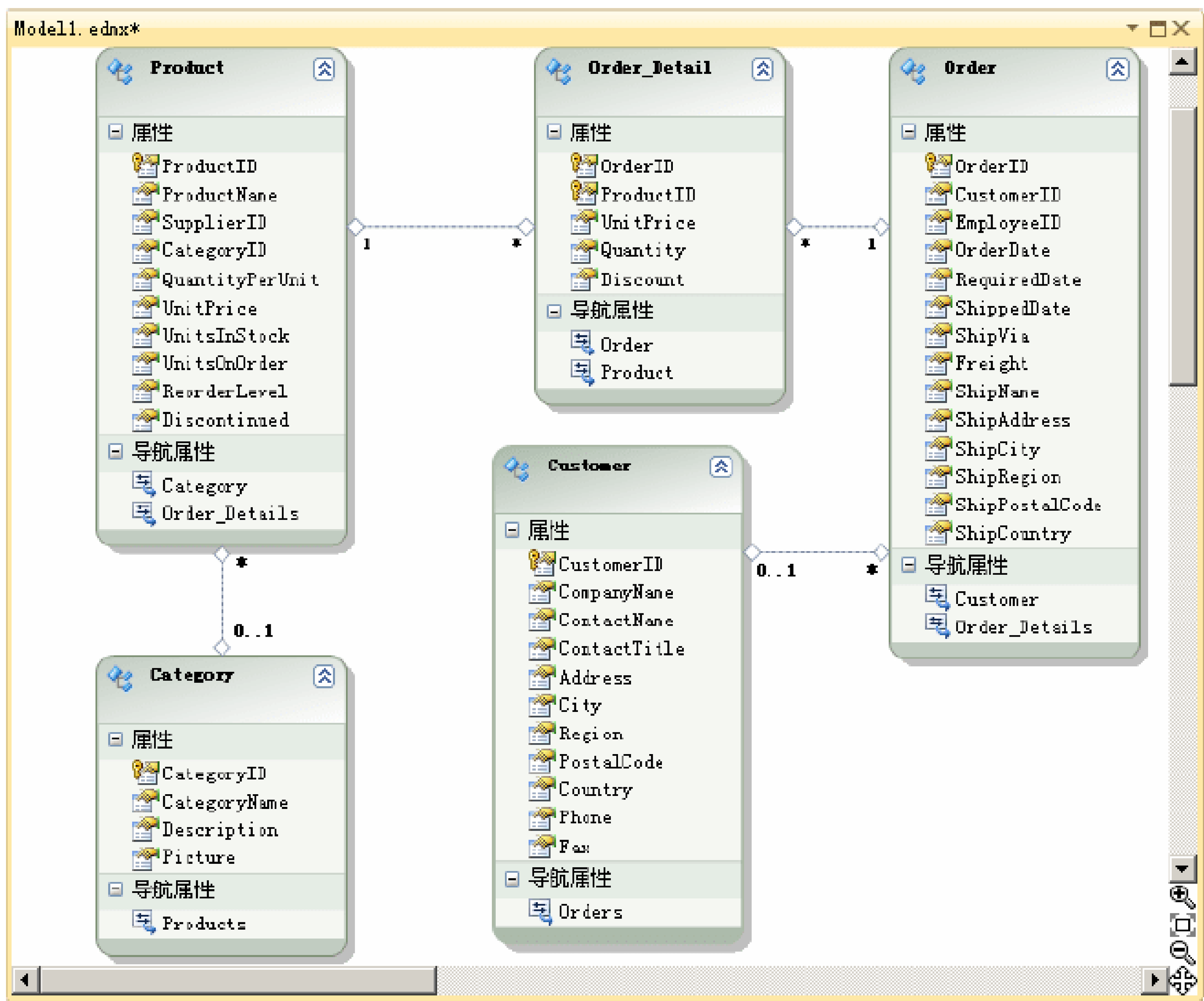


图 8.7　生成的实体模型

从解决方案资源管理器中可以看到，有两个文件与图 8.7 所示的实体模型相关，一个是 Model1.edmx，这是一个 XML 文件，定义了模型中包含的实体、实体之间的关系等，图 8.7 用图形化的方式显示了这个文件的内容。另外一个文件 Model1.designer.cs，这是从

实体模型生成的 C#代码，包括所有实体类的代码、实体之间的关系代码等。

从图 8.7 中可以看出，数据库中的表被映射为实体类，表中字段被映射为实体类的属性，数据库中的外键关系被映射为实体间的导航属性。从后台代码中可以看到，整个实体模型派生自 DataContext 类，而每一个实体类都派生自 EntityObject 类。

```
public partial class NorthwindEntities : ObjectContext
{  ...  }
[EdmEntityTypeAttribute(NamespaceName="NorthwindModel",
Name="Category")]
[Serializable()]
[DataContractAttribute(IsReference=true)]
public partial class Category : EntityObject
{  ...  }
```

8.3.3　查询数据

实体框架模型创建完成后，可以使用 LINQ 语法对实体框架模型中包含的表进行查询，查询得到的结果通常为与数据库对应的实体类，当然也可以在 select 中使用匿名类得到其他类型的查询结果。

【例 8-11】 查询实体框架数据。

本例演示如何使用 LINQ 对实体框架数据进行简单查询。

（1）创建一个 ASP.NET Web 应用程序，在项目中添加前 8.3.2 节所创建的 Northwind 实体框架模型。

（2）在项目中添加一个 SimpleQuery.aspx 页面，页面上放置一个 GridView 控件和一个分页控件 AspNetPager，页面代码如下：

```
<form id="form1" runat="server">
<div>
<h3>简单查询</h3>
<asp:GridView runat="server" id="grid" AutoGenerateColumns="False" Data-
KeyNames="ProductID"  >
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="商品编号"
        ReadOnly="True" />
        <asp:BoundField DataField="ProductName" HeaderText="商品名称"/>
        <asp:BoundField DataField="SupplierID" HeaderText="供货商编号" />
        <asp:BoundField DataField="CategoryID" HeaderText="目录编号" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="包装数
        量" />
        <asp:BoundField DataField="UnitPrice" HeaderText="包装单价" />
        <asp:BoundField DataField="UnitsInStock" HeaderText="库存数量" />
        <asp:BoundField DataField="UnitsOnOrder" HeaderText="订货数量" />
    </Columns>
</asp:GridView>
    <webdiyer:AspNetPager ID="pager1" runat="server"
        onpagechanged="AspNetPager1 PageChanged">
    </webdiyer:AspNetPager>
</div>
</form>
```

（3）在 SimpleQuery.aspx 页面后台代码文件中，添加一个 bindProducts 方法以加载并

显示当前页面的商品数据。

```
private void bindProducts()
{
    NorthwindEntities ef=new NorthwindEntities();
    var query = from p in ef.Products
                orderby p.ProductID
                select p;                           //定义查询
    int count = query.Count();
    pager1.RecordCount = count;                     //设置记录总数
    var list = query
        .Skip((pager1.CurrentPageIndex - 1) * pager1.PageSize)
        .Take(pager1.PageSize)
        .ToList();                                  //取得当前页数据
    grid.DataSource = list;
    grid.DataBind();
}
```

（4）在 SimpleQuery.aspx 页面的 Page_Load 事件和分页控件的 PageChanged 事件中，调用上述 bindProducts()方法以显示最新页面数据。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindProducts();
    }
}
protected void AspNetPager1 PageChanged(object sender, EventArgs e)
{
    bindProducts();
}
```

（5）运行 SimpleQuery.aspx 页面，运行结果如图 8.8 所示。



商品编号	商品名称	供货商编号	目录编号	包装数量	包装单价	库存数量	订货数量
21	Sir Rodney's Scones	8	3	24 pkgs. x 4 pieces	10.0000	3	40
22	Gustaf's Knäckebröd	9	5	24 - 500 g pkgs.	21.0000	104	0
23	Tunnbröd	9	5	12 - 250 g pkgs.	9.0000	61	0
24	Guaraná Fantástica	10	1	12 - 355 ml cans	4.5000	20	0
25	NuNuCa Nuß-Nougat-Creme	11	3	20 - 450 g glasses	14.0000	76	0
26	Gumbär Gummibärchen	11	3	100 - 250 g bags	31.2300	15	0
27	Schoggi Schokolade	11	3	100 - 100 g pieces	43.9000	49	0
28	Rössle Sauerkraut	12	7	25 - 825 g cans	45.6000	26	0
29	Thüringer Rostbratwurst	12	6	50 bags x 30 sausgs.	123.7900	0	0
30	Nord-Ost Matjeshering	13	8	10 - 200 g glasses	25.8900	10	0

图 8.8　实体框架简单查询示例

8.3.4　外键关系和导航属性

在实体框架中，数据库中的外键关系被映射为导航属性。在 Northwind 数据库中，产

品表 Product 的 CategoryId 通过外键关联到类别表 Category 的 CategoryId，这个外键关系在实体框架的模型图中体现为 Product 模型和 Category 模型之间有一条连线，在 Proudct 这一端标记为*，在 Category 这一端标记为 0..1，表示商品可以属于 0 到 1 种类别。同时，在 Product 模型底部的导航属性中有一个 Category 导航属性，通过这个属性可以找到产品所属的类别，在 Cate 模型的导航属性中有一个 Products 导航属性，可以找到该类别的所有商品，如图 8.9 所示。

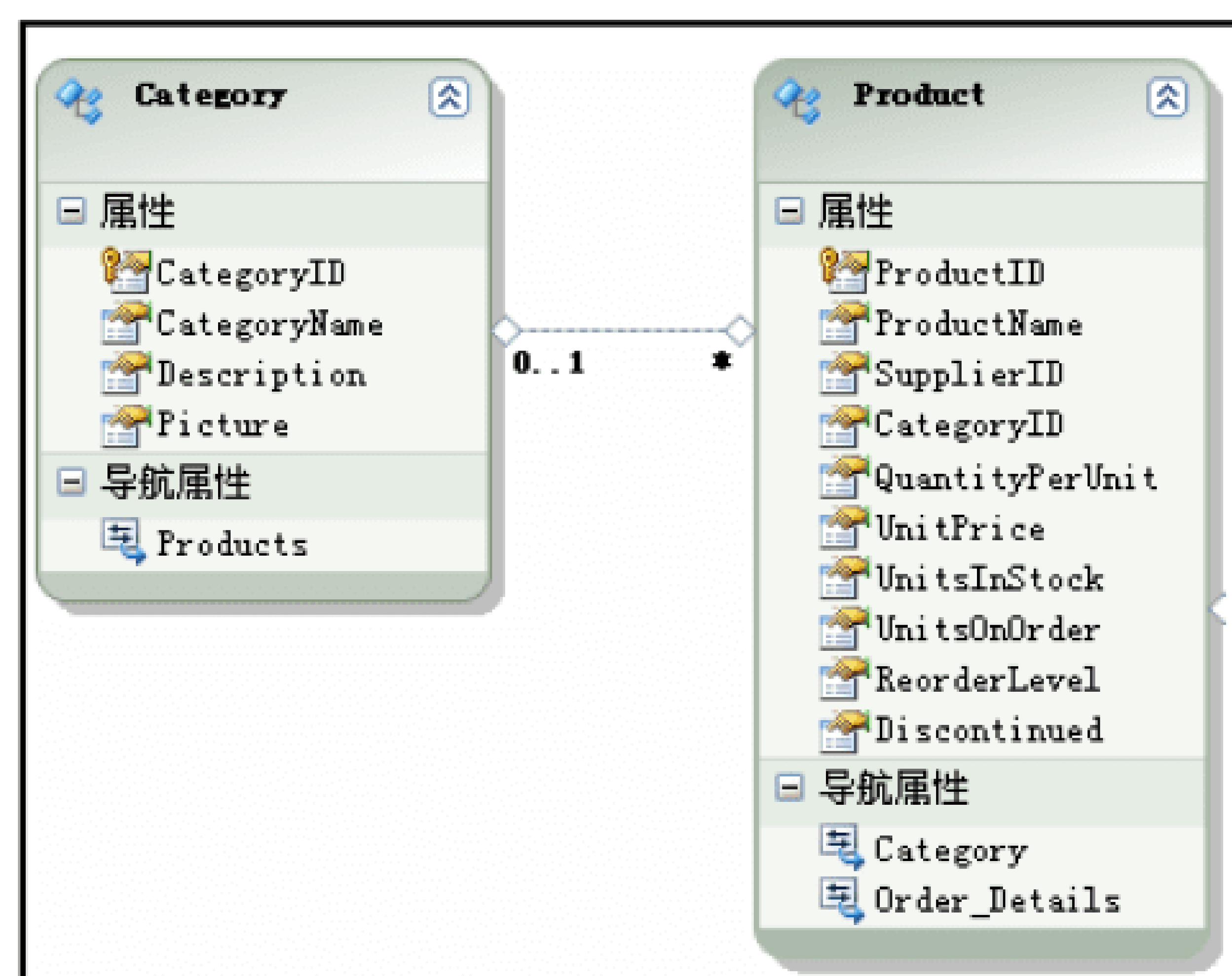


图 8.9　实体框架中的导航属性

用代码来描述图 8.9 所示的导航属性的使用如下：

```
//下面 3 行代码演示通过商品查找类别
Product p= getAProduc t();                          //p 是一个商品
Category c=p.Category;                              //得到商品类别
String categoryName=p.Category.CategoryName;        //得到商品所属类别名称
//下面两行代码演示通过类别查找商品
Category c=getACategory();
IEnumerable<Product> products=c.Products;
```

【例 8-12】 导航属性的使用。

本例演示如何通过导航属性方便的得到外键关系双方数据。

（1）打开例 8-11 所创建的项目，本例将使用其中创建的实体框架模型。

（2）在项目中添加一个页面 NavigationPropertyPage.aspx，在页面上放置一个 GridView 和一个分页控件。

（3）在 GridView 中将显示商品列表，为 GridView 添加商品表的各列，并将商品类别设置为模板列，该列将显示类别名称而非类别编号。利用实体框架的导航属性，通过 Product 类的 Category 属性可以找到商品类别，再访问商品类别的 CategoryName 属性，即可得到商品所属类别名称。页面代码如下：

```
<form id="form1" runat="server">
  <asp:GridView runat="server" id="grid" AutoGenerateColumns="False"
  DataKeyNames="ProductID"  >
   <%--DataGridView 的各个列，共包括 6 个数据绑定列和一个模板列
   <Columns>
       <asp:BoundField DataField="ProductID" HeaderText="商品编号"
       ReadOnly="True" />
       <asp:BoundField DataField="ProductName" HeaderText="商品名称"/>
```

```
        <%--模板列，显示商品类别名称--%>
        <asp:TemplateField HeaderText="商品类别">
        <ItemTemplate>
        <%#Eval("Category.CategoryName")%>
        </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="QuantityPerUnit"HeaderText="包装数量"/>
        <asp:BoundField DataField="UnitPrice" HeaderText="包装单价" />
        <asp:BoundField DataField="UnitsInStock" HeaderText="库存数量" />
        <asp:BoundField DataField="UnitsOnOrder" HeaderText="订货数量" />
    </Columns>
</asp:GridView>
    <%--分页控件--%>
    <webdiyer:AspNetPager ID="pager1" runat="server"
        onpagechanged="AspNetPager1_PageChanged">
    </webdiyer:AspNetPager>
</form>
```

（4）在 NavigationPropertyPage.aspx 页面的 Page_Load 事件和分页控件的 PageChanged 事件中编写代码加载数据，代码如下：

```
public partial class NavigationPropertyPage : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            bindProducts();
        }
    }
    protected void AspNetPager1_PageChanged(object sender, EventArgs e)
    {
        bindProducts();
    }
    private void bindProducts()                                //加载数据
    {
        NorthwindEntities ef = new NorthwindEntities();
        var query = from p in ef.Products
                    orderby p.ProductID
                    select p;                                  //定义查询
        int count = query.Count();
        pager1.RecordCount = count;                            //设置记录总数
        var list = query
            .Skip((pager1.CurrentPageIndex - 1) * pager1.PageSize)
            .Take(pager1.PageSize)
            .ToList();                                         //取得当前页数据
        grid.DataSource = list;
        grid.DataBind();
        IEnumerable<Product> t= list[0].Category.Products;
    }
}
```

（5）运行 NavigationPropertyPage.aspx 页面，运行结果如图 8.10 所示。

（6）本例前几个步骤演示了从外键表到主键表的导航属性，接下再演示从主键表到外键表的导航属性。添加一个新页面 NavigationPropertyPage2.aspx，在页面上放置一个 DropDownList 控件和一个 GridView 控件，DropDownList 用于显示商品类别，GridView 控件用于显示类别下的商品列表。页面代码如下：

```
<%--DropDownList 控件中显示所有类别--%>
<asp:DropDownList runat="server" id="categoryList"
        DataValueField="CategoryId" DataTextField="CategoryName" Auto-
        PostBack=True
        onselectedindexchanged="categoryList SelectedIndexChanged"/>
        <br />
   <asp:GridView runat="server" id="grid" AutoGenerateColumns="False"
   DataKeyNames="ProductID"  >
    <Columns>
    <%--GridView 中包括 6 个数据绑定列--%>
        <asp:BoundField DataField="ProductID" HeaderText="商品编号" Read-
        Only="True" />
        <asp:BoundField DataField="ProductName" HeaderText="商品名称"/>
        <asp:BoundField DataField="QuantityPerUnit"HeaderText="包装数量"/>
        <asp:BoundField DataField="UnitPrice" HeaderText="包装单价" />
        <asp:BoundField DataField="UnitsInStock" HeaderText="库存数量" />
        <asp:BoundField DataField="UnitsOnOrder" HeaderText="订货数量" />
    </Columns>
</asp:GridView>
```



商品编号	商品名称	商品类别	包装数量	包装单价	库存数量	订货数量
1	product!!	Beverages	10 boxes x 20 bags	18.0000	39	0
2	Chang	Beverages	24 - 12 oz bottles	19.0000	17	40
3	Aniseed Syrup	Condiments	12 - 550 ml bottles	10.0000	13	70
4	Chef Anton's Cajun Seasoning	Condiments	48 - 6 oz jars	22.0000	53	0
5	Chef Anton's Gumbo Mix	Condiments	36 boxes	21.3500	0	0
6	Grandma's Boysenberry Spread	Condiments	12 - 8 oz jars	25.0000	120	0
7	Uncle Bob's Organic Dried Pears	Produce	12 - 1 lb pkgs.	30.0000	15	0
8	Northwoods Cranberry Sauce	Condiments	12 - 12 oz jars	40.0000	6	0
9	Mishi Kobe Niku	Meat/Poultry	18 - 500 g pkgs.	97.0000	29	0
10	Ikura	Seafood	12 - 200 ml jars	31.0000	31	0

图 8.10　导航属性示例 1

（7）在 NavigationPropertyPage2.aspx 页面的 PageLoad 事件中，绑定所有类别到 DropDownList 控件。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
         NorthwindEntities ef = new NorthwindEntities();
         categoryList.DataSource = ef.Categories.ToList();   //得到所有类别
         categoryList.DataBind();
    }
}
```

（8）在 DropDownList 的 SelectedIndexChanged 事件中，显示当前选中类别下的所有商品列表，代码如下：

```
protected void categoryList SelectedIndexChanged(object sender, EventArgs
e)
{
    NorthwindEntities ef = new NorthwindEntities();
```

```
    int n = int.Parse(categoryList.SelectedValue);//得到所选择的类别 ID
    Category category = (from c in ef.Categories
                         where c.CategoryID == n
                         select c
                         ).Single();                  //根据 ID 得到类别信息
    var list = category.Products;                     //得到类别下的所有商品
    grid.DataSource = list;
    grid.DataBind();
}
```

（9）运行 NavigationPropertyPage2.aspx 页面，运行界面如图 8.11 所示。



商品编号	商品名称	包装数量	包装单价	库存数量	订货数量
11	Queso Cabrales	1 kg pkg.	21.0000	22	30
12	Queso Manchego La Pastora	10 - 500 g pkgs.	38.0000	86	0
31	Gorgonzola Telino	12 - 100 g pkgs	12.5000	0	70
32	Mascarpone Fabioli	24 - 200 g pkgs.	32.0000	9	40
33	Geitost	500 g	2.5000	112	0
59	Raclette Courdavault	5 kg pkg.	55.0000	79	0
60	Camembert Pierrot	15 - 300 g rounds	34.0000	19	0
69	Gudbrandsdalsost	10 kg pkg.	36.0000	26	0
71	Flotemysost	10 - 500 g pkgs.	21.5000	26	0
72	Mozzarella di Giovanni	24 - 200 g pkgs.	34.8000	14	0

图 8.11　导航属性示例 2

8.3.5　修改数据

利用实体框架，不但可以方便地检索数据，而且很容易实现添加、删除和修改数据。本节将通过代码示例的方式说明如何添加、删除和修改数据，在后面的内容中将介绍这种机制的原理。

【例 8-13】 添加、删除、修改数据。

本例演示如何通过实体框架实现添加、删除、修改商品信息。

（1）打开例 8-11 所创建的项目，本例将使用其中创建的实体框架模型。

（2）在项目中添加一个页面 EditProductPage.aspx，用以修改商品信息。页面上包括数个 TextBox 控件和一个 DropDownList 控件，DropDownList 用于显示商品所属类别，TextBox 控件用于显示商品其他属性。页面顶部放置 3 个 Button 分别实现更新、删除和添加功能。EditProductPage.aspx 页面代码如下：

```
<form id="form1" runat="server">
<asp:Button runat="server" Text="更新" id="updateButton" />
<asp:Button runat="server" Text="删除" id="deleteButton" />
<asp:Button runat="server" Text="添加" id="addButton" />
<table>
    <tr>
    <td>商品编号：</td>
```

```
    <td><asp:TextBox runat="server" id="productId" ReadOnly="True"/>
    </asp:TextBox></td>
    </tr>
    <tr>
    <td>商品名称:</td>
    <td><asp:TextBox runat="server" id="productName" /></asp:TextBox>
    </td>
    </tr>
    <tr>
    <td>所属类别:</td>
    <td><asp:DropDownList runat="server" id="category" DataValueField=
    "CategoryId" DataTextField="CategoryName" /></asp:DropDownList></td>
    </tr>
    <tr>
    <td>包装数量:</td>
    <td><asp:TextBox runat="server" id="unitQuantity" /></asp:TextBox>
    </td>
    </tr>
    <tr>
    <td>包装单价:</td>
    <td><asp:TextBox runat="server" id="unitPrice" /></asp:TextBox></td>
    </tr>
    <tr>
        <td>库存数量:</td>
    <td><asp:TextBox runat="server" id="stockQuantity" /></asp:TextBox>
    </td>
    </tr>
    <tr>
    <td>已定货量:</td>
    <td> <asp:TextBox runat="server" id="orderUnits"></asp:TextBox>
    </td>
    </tr>
    <tr>
    <td>再定货点:</td>
    <td> <asp:TextBox runat="server" id="reorderLevel"></asp:TextBox>
    </td>
    </tr>
    <tr>
    <td>停止供货:</td>
    <td><asp:CheckBox runat="server" id="discontinued"></asp:CheckBox>
    </td>
    </tr>
</table>
</form>
```

（3）在 Page_Load 事件中，根据 QueryString 所传递的商品 ID 值，从数据库中检索对应商品信息并显示在页面上。

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        loadData();
        //绑定商品类别列表
        NorthwindEntities ef = new NorthwindEntities();
        category.DataSource = ef.Categories;
        category.DataBind();
    }
}
```

```
//根据商品 ID 从数据库加载商品信息并显示
private void loadData()
{
    string s = Request.QueryString["id"];      //从 QueryString 获得商品 ID
    if (string.IsNullOrEmpty(s)) return;
    int id = int.Parse(s);
    NorthwindEntities ef = new NorthwindEntities();
    //根据 ID 查找商品
    Product product = (from p in ef.Products
                       where p.ProductID == id
                       select p).SingleOrDefault();
    if (product == null)
    {
        productName.Text = "[未找到这个产品]";
        return;
    }
    //将此商品各个属性显示在相应控件中
    productId.Text = id.ToString();
    productName.Text = product.ProductName;
    category.SelectedValue = product.CategoryID.ToString();
    unitQuantity.Text = product.QuantityPerUnit.ToString();
    unitPrice.Text = product.UnitPrice.ToString();
    stockQuantity.Text = product.UnitsInStock.ToString();
    orderUnits.Text = product.UnitsOnOrder.ToString();
    reorderLevel.Text = product.ReorderLevel.ToString();
    discontinued.Checked = product.Discontinued;
    ef.Dispose();
}
```

（4）在“更新”按钮的 Click 事件中，将用户编辑后的商品信息保存到数据库中。注意代码中通过调用 NorthwindEntities 类的 SaveChanges 方法实现数据保存功能。

```
protected void updateButton Click(object sender, EventArgs e)
{
    NorthwindEntities ef = new NorthwindEntities();
    int id=Convert.ToInt32(productId.Text);
    //根据商品 ID 查找商品
    var product = (from p in ef.Products
                  where p.ProductID == id
                  select p)
                  .SingleOrDefault();
    //如果商品未找到则提示错误
    if (product == null)
    {
        Page.ClientScript.RegisterStartupScript(this.GetType(), "showm-
        essage", "<script>alert('数据库中不存在此 id 所对应的商品，不能更新。');
        </script>");
        return;
    }
    //逐个设置商品的各个属性
    product.ProductName = productName.Text;
    product.CategoryID = int.Parse(category.SelectedValue);
    product.QuantityPerUnit = unitQuantity.Text;
    product.UnitPrice = decimal.Parse(unitPrice.Text);
    product.UnitsInStock = short.Parse(stockQuantity.Text);
    product.UnitsOnOrder = short.Parse(orderUnits.Text);
    product.ReorderLevel = short.Parse(reorderLevel.Text);
    product.Discontinued = discontinued.Checked;
```

```
    //将变化保存到数据库
    ef.SaveChanges();
    ef.Dispose();
    Page.ClientScript.RegisterStartupScript(this.GetType(), "showmessa-
    ge",
        "<script>alert('商品信息已经保存到数据库');</script>");
}
```

（5）在“删除”按钮的 Click 事件中，根据商品 ID 删除当前商品。

```
protected void deleteButton_Click(object sender, EventArgs e)
{
    NorthwindEntities ef = new NorthwindEntities();
    int id = Convert.ToInt32(productId.Text);
    //根据商品 ID 查找商品
    var product = (from p in ef.Products
                   where p.ProductID == id
                   select p)
                  .SingleOrDefault();
    //如果商品未找到则提示错误
    if (product == null)
    {
        Page.ClientScript.RegisterStartupScript(this.GetType(), "showm-
        essage", "<script>alert('数据库中不存在此 ID 所对应的商品，不能更新。');
        </script>");
        return;
    }
    //从实体框架上下文删除商品并将变化提交到数据库
    ef.DeleteObject(product);
    ef.SaveChanges();
    ef.Dispose();
    Page.ClientScript.RegisterStartupScript(this.GetType(), "showmessa-
    ge", "<script>alert('商品已经从数据库中删除。');</script>");
}
```

（6）在“添加”按钮的 Click 事件中，将用户输入的新商品信息保存到数据库。

```
protected void addButton Click(object sender, EventArgs e)
{
    //创建一个新产品实体并根据用户输入设置各个属性
    Product product = new Product();
    product.ProductID = int.Parse(productId.Text);
    product.ProductName = productName.Text;
    product.CategoryID = int.Parse(category.SelectedValue);
    product.QuantityPerUnit = unitQuantity.Text;
    product.UnitPrice = decimal.Parse(unitPrice.Text);
    product.UnitsInStock = short.Parse(stockQuantity.Text);
    product.UnitsOnOrder = short.Parse(orderUnits.Text);
    product.ReorderLevel = short.Parse(reorderLevel.Text);
    product.Discontinued = discontinued.Checked;
    //将新产品添加到实体框架上下文
    NorthwindEntities ef = new NorthwindEntities();
    ef.Products.AddObject(product);
    //将变化保存到数据库
    ef.SaveChanges();
    ef.Dispose();
```

```
    Page.ClientScript.RegisterStartupScript(this.GetType(), "show-
    message", "<script>alert('新商品信息已经添加到数据库');</script>");
}
```

（7）运行 EditProductPage.aspx 页面，运行界面如图 8.12 所示。

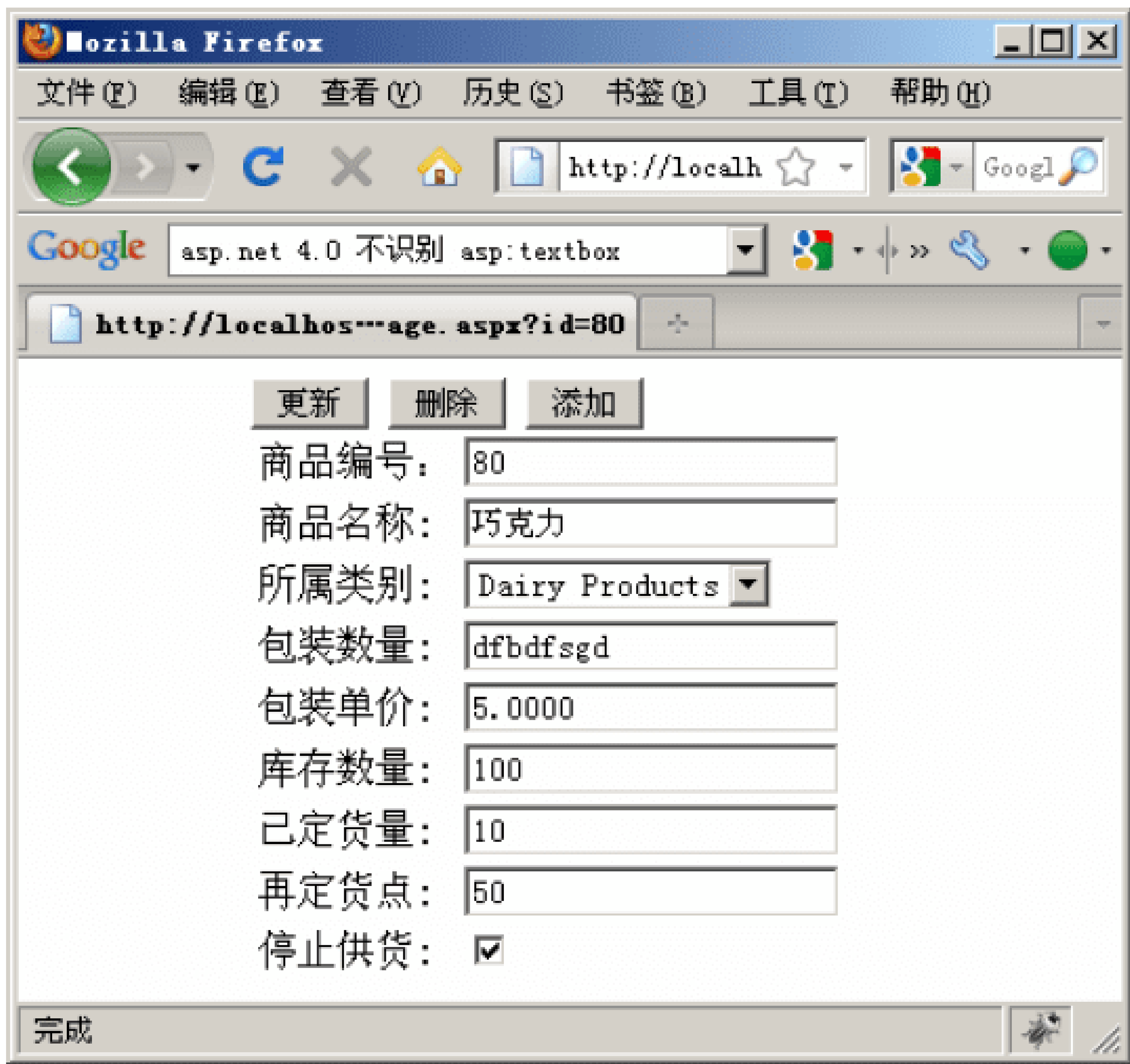


图 8.12　商品修改页面运行界面

8.4　深入理解实体框架

8.3 节中介绍了实体框架的基本概念和基本操作，通过前面的例子可以看出，使用实体框架只需编写少量代码就可以实现数据增删改查。相对于开发人员编写的简单代码来说，实体框架本身却是一个很复杂的框架，封装了对象映射、状态维持、连接管理、SQL 语句生成等众多功能。正是由于实体框架对这些功能的全面管理和封装，才使得开发人员编写数据访问代码更为简单。深入理解实体框架的运行机制对于开发人员编写高效的、功能复杂的数据访问代码来说是一个必要条件。

8.4.1　对象上下文 ObjectContext

对象上下文 ObjectContext 类是以对象（这些对象是 EDM 中定义的实体类型的实例）的形式与数据进行交互的主要类。ObjectContext 类的实例封装以下内容：

- 到数据库的连接，以 EntityConnection 对象的形式封装。
- 描述模型的元数据，以 MetadataWorkspace 对象的形式封装。
- 在创建、更新和删除操作过程中跟踪对象的 ObjectStateManager 对象。

在前面的例子中，NorthwindEntities 类就是一个对象上下文，继承自 ObjectContext 类，可以从自动生成的代码中看到这一点。ObjectContext 被创建时，需要指定其连接字符串和实体容器名称，实体容器名称必须是唯一的，可以在实体模型设计视图的属性窗口中修改这个名称。下面是自动生成的 NorthwindEntities 类的一部分代码，其中包括了类继承层次和构造函数。

```
public partial class NorthwindEntities : ObjectContext
{
    #region 构造函数
    //使用应用程序配置文件中名为 NorthwindEntities 的连接字符串初始化
    public NorthwindEntities() : base("name=NorthwindEntities", "North-
    windEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;     //启动延迟加载
        OnContextCreated();
    }
    //初始化新的 NorthwindEntities 对象
    public NorthwindEntities(string connectionString) : base(connectionS-
    tring, "NorthwindEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }
    //初始化新的 NorthwindEntities 对象
    public NorthwindEntities(EntityConnection connection) : base(conn-
    ection, "NorthwindEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }
    #endregion
}
```

一个实体对象只有在对象上下文中时，才能被更新、删除和修改。这一点从 8.3 节的例子中也可以看得出。在 8.3 节对商品数据进行修改的例子中，先对商品实体类 Product 的实例进行修改或者删除，最后调用对象上下文类 NorthwindEntities 的 SaveChanges 方法将修改保存到数据库。如果一个对象不在对象上下文中，则无法将其修改提交到数据库。

实体对象既可以在一个对象上下文中，也可以不在，这两种状态之间也可以相互转换。将一个孤立的实体对象添加到对象上下文中称为附加，反之，使一个对象上下文中的实体对象脱离对象上下文称为分离。在实体框架的某个对象上下文内执行查询时，返回的对象会自动附加到该对象上下文。如 8.3 节所执行的查询操作都是通过 NorthwindEntities 对象上下文进行的，所以查询到的实体对象自动附加到该对象上下文。而使用 new 关键字创建的实体对象并不属于任何对象上下文。如下面的例子所示。

```
static void Main(string[] args)
{
    NorthwindEntities northwind = new NorthwindEntities();
    //p1 是从对象上下文 northwind 中查询得到，所以属于 northwind
    var p1 = (from p in northwind.Products
              where p.ProductID==1
                  select p)
                .First();
    //对象 p2 使用构造函数单独创建，不属于任何对象上下文
    Product p2 = new Product() { ProductID=1, CategoryID = 1, ProductName
    = "测试" };
    p1.ProductName = "新名称 1";
    p2.ProductName = "新名称 2";
    //调用对象上下文的 SaveChanges 方法时，只有 p1 的改变会更新到数据库
    northwind.SaveChanges();
    northwind.Dispose();
```

```
}
```

如果一个实体对象不属于任何对象上下文，则可以使用使用下列方法之一将对象附加到对象上下文：

- 调用 ObjectContext 上的 AddObject 将对象附加到对象上下文。当对象为数据源中尚不存在的新对象时采用此方法。
- 调用 ObjectContext 上的 Attach 将对象附加到对象上下文。当对象已存在于数据源中但当前尚未附加到上下文时采用此方法。
- 调用 ObjectContex 的 AttachTo，以将对象附加到对象上下文中的特定实体集。
- 调用 ObjectContext 上的 ApplyPropertyChanges 方法。当对象已存在于数据源中，并且分离的对象具有希望保存的属性更新时采用此方法。如果简单地附加该对象，则属性更改将丢失。

在 8.3 节添加商品信息的例子中，就是通过 Add 方法将新创建的 Product 对象添加到对象上下文，如下代码所示。

```
protected void addButton_Click(object sender, EventArgs e)
{
    //创建一个新产品实体并根据用户输入设置各个属性
    Product product = new Product();
    product.ProductID = int.Parse(productId.Text);
    product.ProductName = productName.Text;
    product.CategoryID = int.Parse(category.SelectedValue);
    product.QuantityPerUnit = unitQuantity.Text;
    product.UnitPrice = decimal.Parse(unitPrice.Text);
    product.UnitsInStock = short.Parse(stockQuantity.Text);
    product.UnitsOnOrder = short.Parse(orderUnits.Text);
    product.ReorderLevel = short.Parse(reorderLevel.Text);
    product.Discontinued = discontinued.Checked;
    //将新产品添加到实体框架上下文
    NorthwindEntities ef = new NorthwindEntities();
    ef.Products.AddObject(product);
    //将变化保存到数据库
    ef.SaveChanges();
    ef.Dispose();
    Page.ClientScript.RegisterStartupScript(this.GetType(), "showm-
    essage", "<script>alert('新商品信息已经添加到数据库');</script>");
}
```

将对象附加到对象上下文时应注意以下注意事项：

- 如果被附加的对象具有相关对象，则这些对象也被附加到对象上下文。
- 若要使用 Attach 附加一个对象，该对象必须实现 IEntityWithKey 并具有有效的键。
- 对象以 Unchanged 状态附加到对象上下文。
- 如果附加的对象不在数据源中，则在执行 SaveChanges 时不会添加该对象。在这种情况下，如果对属性进行了更改，则在执行 SaveChanges 时在服务器上会引发异常。若要添加对象，应该使用 Add()方法而不是 Attach()方法。

如果要将一个当前存在于对象上下文中的对象分离，则可以调用 ObjectContext 类的 Detach()方法，如下代码所示。

```
NorthwindEntities northwind = new NorthwindEntities();
//p1 是从对象上下文 northwind 中查询得到，所以属于 northwind
```

```
var p1 = (from p in northwind.Products
         where p.ProductID==1
              select p)
           .First();
//将 p1 从 northwind 上下文分离
northwind.Detach(p1);
```

8.4.2　对象状态和对象修改

在实体框架中，实体对象有以下几种状态：

- ❑ Detached 状态：对象存在，但没有被对象服务跟踪。实体在创建之后且添加到对象上下文之前处于此状态。通过调用 Detach 方法从上下文中移除或者如果使用 NoTrackingMergeOption 加载，实体也处于此状态。
- ❑ Unchanged 状态：自加载到对象上下文中后，或自上次调用 SaveChanges 方法后，对象尚未修改。
- ❑ Added 状态：对象添加到对象上下文，SaveChanges 方法尚未调用。通过调用 AddObject 或者 Add()方法将对象添加到对象上下文。
- ❑ Deleted 状态：通过使用 DeleteObject()方法从对象上下文删除对象后处于此状态。
- ❑ Modified 状态：对象已更改，但尚未调用 SaveChanges()方法。

当调用 ObjectContext 类的 SaveChanges()方法时，实体框架会根据对象的状态生成对应的数据操作语句并执行。如果对象处于 Added 状态，则生成一条 INSERT 语句，如果对象处于 Deleted 状态，则生成一条 DELETE 语句，如果对象处于 Modified 状态，则生成一条 UPDATE 语句。如果对象处于 Unchanged 状态，则不需要处理。如果对象处于 Detached 状态，即未附加到对象上下文，则也不会对此对象进行处理。可以通过 SQL Server 提供的监视工具 SQL Server Profiler 来查看 Entity Framework 生成的 SQL 语句。

提示： 由于 Entity Framework 需要将 C#的 LINQ 查询语法转换成对应的 SQL 语句，所以有很多 C#方法由于缺少对应的 SQL 函数而不能在 LINQ 中使用。例如，以下查询语句 from p in northwind.Products where p.ID=int.Parse(textBox1.Text) select p;在执行时就会抛出异常，因为 Entity Framework 无法将 int.Parse 方法无法转换成对应的 SQL 函数。

【例 8-14】 通过 SQL Server Profiler 查看 Entity Framework 生成的代码。

（1）创建一个控制台应用程序，并从 Northwind 数据库生成实体数据模型。

（2）在 Main()方法中编写代码，对产品进行添加、删除和修改。

```
static void Main(string[] args)
{
    NorthwindEntities context = new NorthwindEntities();
                                                    //创建 ObjectContext
    //从 ObjectContext 查找一个商品
    var query = from p in context.Products
                where p.ProductID == 1
                select p;
    Product product = query.First();
    //修改商品并保存到数据库
    product.ProductName = "新的商品名称";
```

```
    context.SaveChanges();
    //创建一个新商品并保存到数据库
    Product newProduct = new Product();
    newProduct.ProductName = "新添加的商品";
    newProduct.CategoryID = 1;
    newProduct.UnitPrice = 20;
    context.Products.AddObject(newProduct);
    context.SaveChanges();
    //删除刚才添加的新商品
    context.Products.DeleteObject(newProduct);
    context.SaveChanges();
}
```

（3）启动 SQL Server Profiler 以监视 SQL 语句。从 Windows 程序菜单中选择 Microsoft SQL Server 2005 | “性能工具” | SQL Server Profiler，则打开如图 8.13 所示的 SQL Server Profiler 主界面。

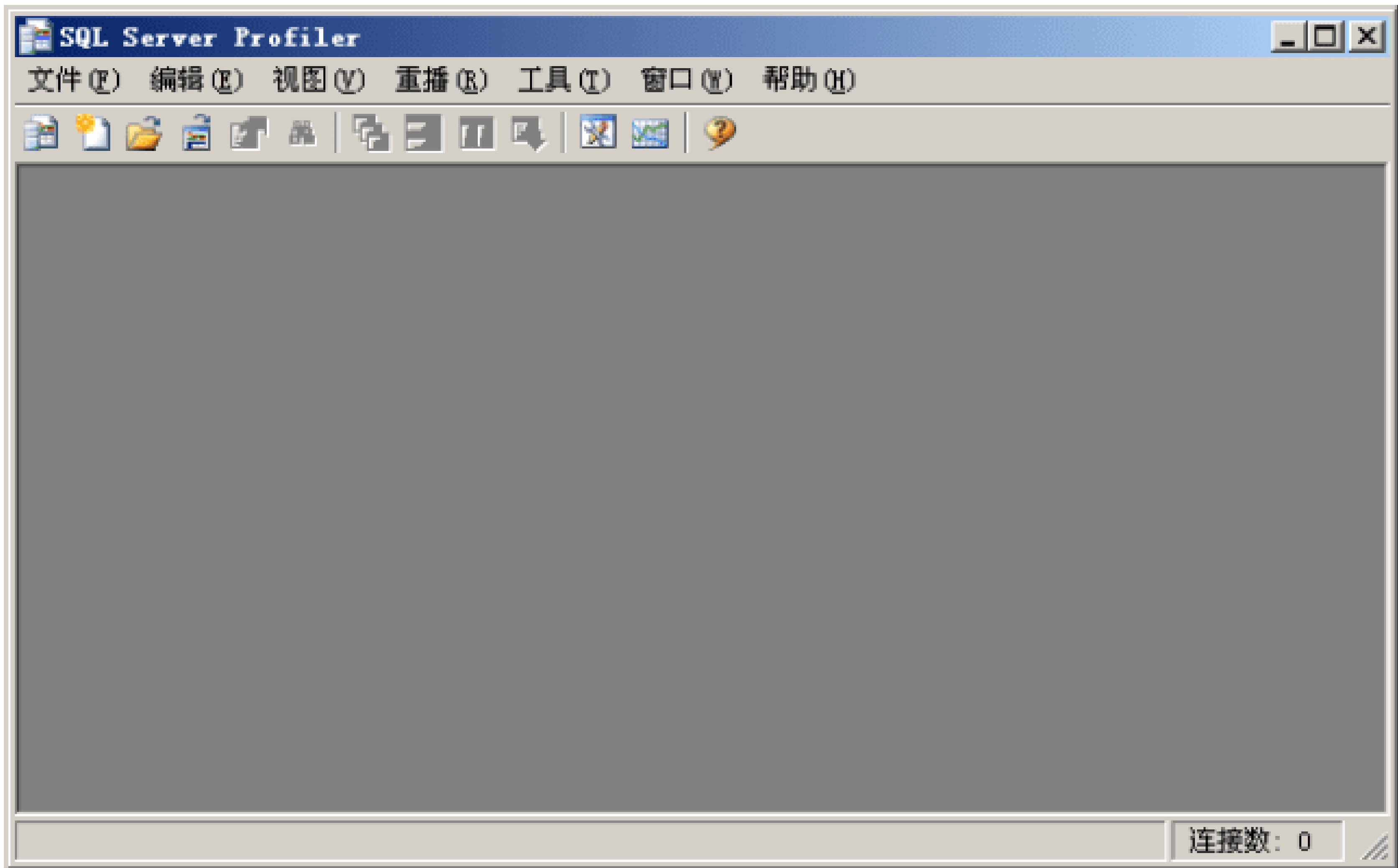


图 8.13　SQL Server Profiler 主界面

（4）在 SQL Server Profiler 主界面中，从菜单中选择“文件” | “新建跟踪”命令，在打开的连接对话框中设置数据库连接，然后进入如图 8.14 所示的“跟踪属性”对话框。在其中可以设置需要监视的 SQL 事件（如建立连接、执行存储过程、执行 SQL 语句等），本例中保存默认值即可，单击对话框底部的“运行”按钮以启动监视。

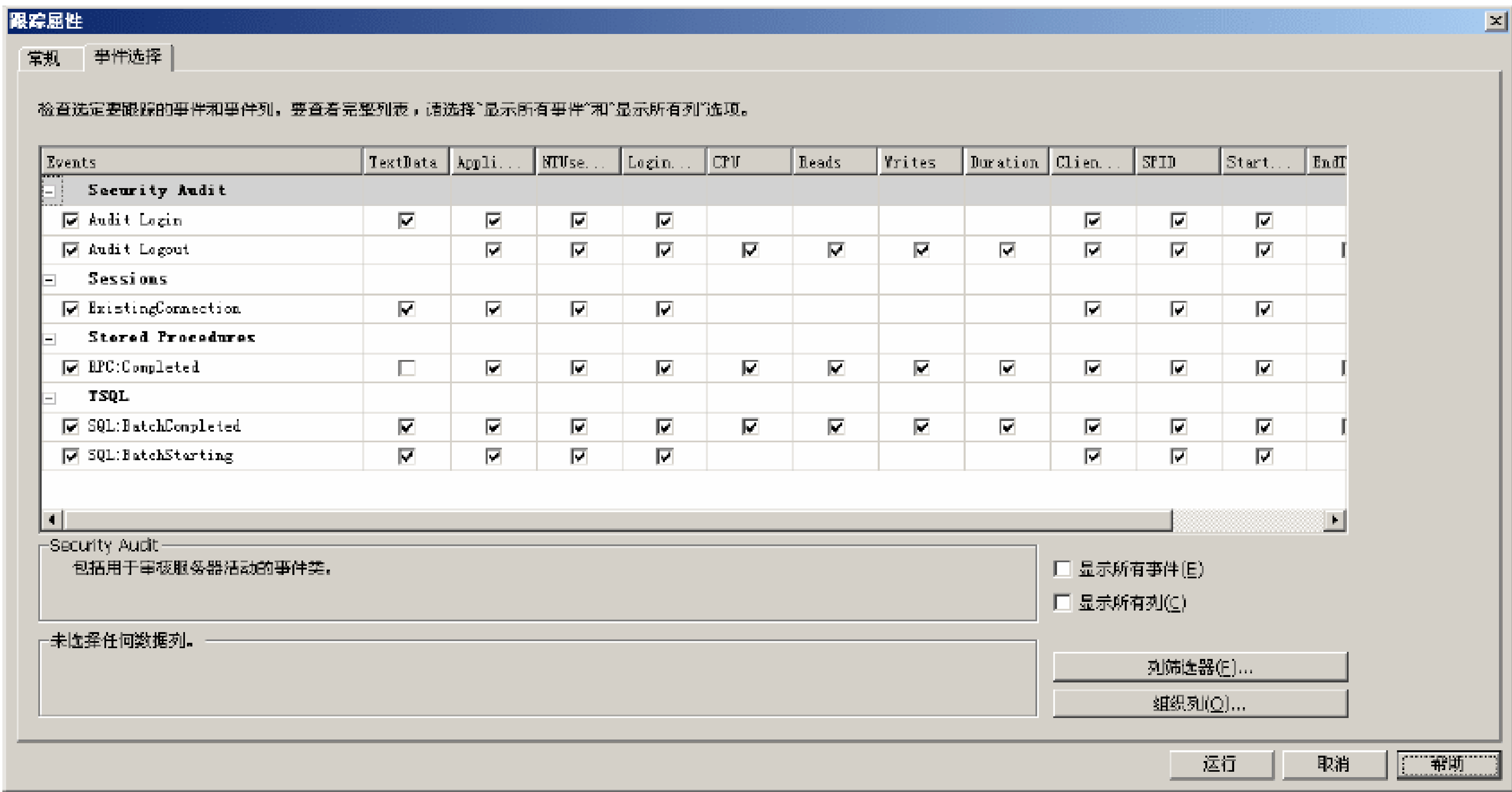


图 8.14　设置跟踪属性

（5）启动 SQL Server Profiler 监视以后，运行刚才所编写的 C#代码，并注意观察 SQL Server Profiler 的输出，通过单步执行 C#代码并同时对比 SQL Server Profiler 的输出即可找出 Entity Framework 的数据操作所生成的 SQL 代码。如图 8.15 所示为 Entity Framework 生成的添加商品的 INSERT 语句。

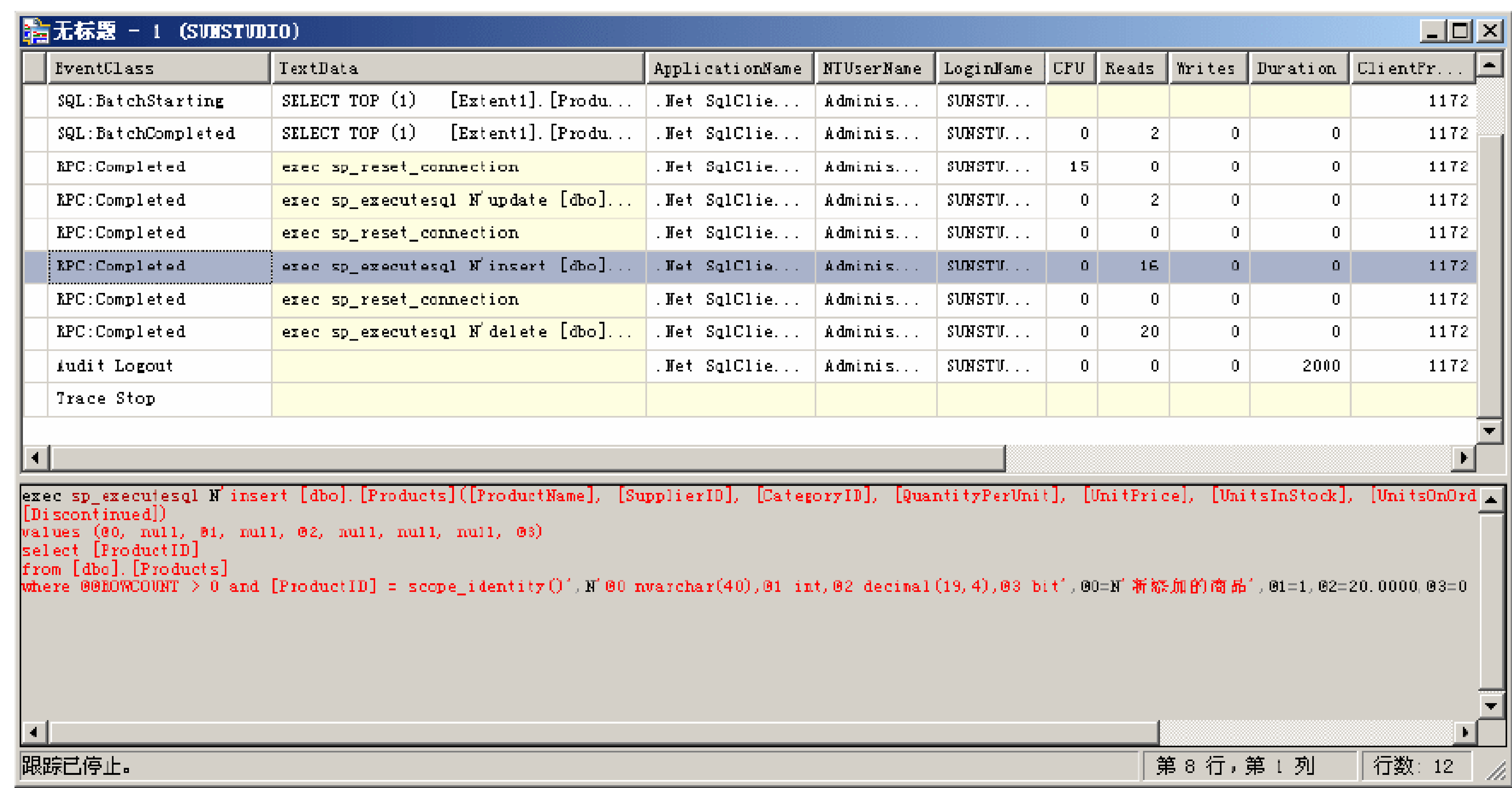


图 8.15　SQL Server Profiler 运行界面

8.5　小　　结

本章介绍了.NET Framework 中最新的数据访问技术：LINQ to Entity Framework。首先介绍了 LINQ 的基本操作，然后介绍了从数据库生成实体对象模型并检索和更新数据，最后对 Entity Framework 的运行机制进行了简单讲解。

第 9 章　ASP.NET AJAX 框架

在现在的 Web 应用中，AJAX 技术以其优秀的性能和良好的用户体验而获得了人们的喜爱，得到了广泛的应用。AJAX 的全称是 Asynchronous JavaScript and XML（异步 JavaScript 和 XML），AJAX 本身并不是一项新技术，而是多种现有技术的综合运用。随着 AJAX 技术的流行，出现了许多 AJAX 框架，目前在 ASP.NET 平台上开发 AJAX 应用主要有两种框架：ASP.NET AJAX 和 jQuery。本章和第 10 章将分别对这两种框架进行介绍。

9.1　AJAX 原理

虽然使用 AJAX 框架进行简单的开发不需要理解 AJAX 运行原理，但是从提高开发人员编程水平和设计思想的角度来说，理解 AJAX 原理是很有必要的。而且，如果某些情况下不允许使用 AJAX 框架，而是完全需要自己写代码实现 AJAX，那么 AJAX 原理就是必须要掌握的一个内容。

9.1.1　AJAX 的意义

在传统的 Web 应用程序中，当用户单击页面上的一个按钮时，通常会引起页面的提交，页面提交到服务器端后，服务器端对其进行处理，然后再返回一个页面给浏览器，浏览器加载并显示这个页面。从用户的角度来看，单击了页面一个按钮，页面变成空白，等待若干时间，页面重新加载并显示。这种传统的 Web 页面有以下两个特点：

（1）同步。从用户提交数据到重新加载新的页面整个过程是同步进行的。服务器在处理完提交数据并返回新页面以前，浏览器端只能处于等待状态，看不到任何结果，也不能进行任何操作。这种同步性使得用户等待时间增加。

（2）整页提交和整页返回。页面提交是需要提交整个页面，哪怕用户只修改了页面上的一个小数据。服务器返回页面时也是需要返回整个页面，哪怕页面上只有一小部分需要更新。这种整页提交和返回使得网络传输量增大，而通常情况下这种数据传输并不是必要的。

AJAX 技术的出现很好地解决了上述两个问题。AJAX 采用异步方式与服务器端交互，避免了用户长时间等待；采用页面局部刷新技术，不必提交和返回整个页面，降低网络流量，节省网络带宽，提高下载速度。

9.1.2　XMLHttpRequest 对象

在浏览器端使用 XMLHttpRequest 对象与服务器代码进行交互。XMLHttpRequest 对象可以向服务器发送一个 HTTP 请求（GET 或者 POST），并取得服务器的响应结果。利用 XMLHttpRequest 对象的这个功能，在浏览器端使用 JavaScript 代码通过 XMLHttpRequest 向服务器发送请求，取得响应后，用 JavaScript 显示在页面上，从而实现了异步、局部刷新页面的效果。

XMLHttpRequest 对象并没有完全统一的标准，在各个不同浏览器或者同一浏览器的不同版本中，创建和使用 XMLHttpRequest 对象也有所不同。在 IE 7 以前的版本中，需要使用以下两种语法之一创建一个 XMLHttpRequest 对象（具体使用哪种语法取决于 IE 版本）。

```
var request=new ActiveXObject("Msxml2.XMLHTTP");
var request=new ActiveXObject("Microsoft.XMLHTTP");
```

在 IE 7 以上版本或者除 IE 以外的多数主流浏览器（如 FireFox 等）中，都使用 XMLHttpRequest 的构造函数创建，如下代码所示。

```
var request=new XMLHttpRequest();
```

为了使编写的代码兼容各种常见浏览器（包括 IE 6），应在创建 XMLHttpRequest 对象前检测浏览器是否支持所调用的方法。可以编写一个辅助类及一组方法来封装浏览器兼容性检测及创建 XMLHttpRequest 对象的功能，如下代码所示。

```
<script type="text/javascript">
    //HTTP 这个对象封装了创建 XMLHttpRequest 的功能
    HTTP = {};
    //定义 3 个工厂函数，分别调用 3 种不同方式创建 XMLHttpRequest
    HTTP._factories = [
function () { return new XMLHttpRequest(); },
function () { return new ActiveXObject("Msxml2.XMLHTTP"); },
function () { return new ActiveXObject("Microsoft.XMLHTTP"); }
];
HTTP._factory = null;                        //适用于当前浏览器的工厂函数
HTTP.newRequest = function () {
    if (HTTP._factory != null)               //如果已经找到合适的函数，直接返回
        return HTTP._factory();
    //循环检测每个工厂函数，直到成功创建一个 XMLHttpRequest 对象
    for (var i = 0; i < HTTP._factories.length; i++) {
        try {
            var factory = HTTP._factories[i];
            var request = factory();
            if (request != null) {
                HTTP._factory = factory;
                return request;
            }
        }
        catch (e) { continue; }
    }
    //如果未找到合适的工厂函数，则抛出异常
    HTTP._factory = function () {
        throw new Error("当前浏览器不支持 XMLHttpRequest.");
    };
```

```
    HTTP._factories();
}
</script>
```

XMLHttpRequest 对象有以下主要属性。

- onreadystatechange：请求状态发生变化时的事件处理程序。
- readyState：请求的当前状态。
- responseBody：以字节数组形式表示的服务器返回内容。
- responseText：以文本形式表示的服务器返回内容。
- responseXML：服务器返回的 XML 文档。
- status：服务器返回的 HTTP 状态代码。
- statusText：服务器返回的 HTTP 状态文本。

XMLHttpRequest 对象主要有以下方法。

- abort：取消当前请求。
- open：打开一个请求，设置请求的方法（GET 或 POST）、目的 URL 等参数。
- send：发送 HTTP 请求。
- setRequestHeader：设置 HTTP 请求头。

9.1.3　一个简单的 AJAX 例子

前面介绍了 AJAX 的相关理论知识，下面将通过一个具体的例子来说明如何编写代码使用 AJAX。

【例 9-1】 利用 AJAX 取得服务器当前时间。

本例演示通过 AJAX 技术异步无刷新地取得服务器端时间。为了更好地说明 AJAX 技术与普通 Web 技术的区别，本例包括两个页面：一个不采用 AJAX 技术的页面和一个采用 AJAX 技术的页面，将二者运行结果作对比，能够更直观地看到 AJAX 的优势。

（1）创建一个 HTTP Web 应用程序，在项目中添加一个页面 TimePage.aspx，页面上放置一个 Label 和一个 Button，Label 用于显示时间，Button 用于从服务器获取时间。页面代码如下：

```
服务器当前时间为：<asp:Label runat="server" id="label1" Text="Label"></asp:
Label><br />
<asp:Button runat="server" id="ok" Text="刷新" onclick="ok Click" />
// "刷新" 按钮的 Click 事件代码如下
protected void ok Click(object sender, EventArgs e)
{
    //为了使页面延迟更加明显，线程休眠一定时间
    Thread.Sleep(500);
    label1.Text = DateTime.Now.ToString("T");
}
```

（2）运行 TimePage.aspx，可以看到第一次单击按钮，页面都需要等待一段时间。

（3）在项目中添加一个一般事件处理程序 HttpHandler，用于处理 AJAX 请求。

```
public class TimeHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
```

```
    {
        context.Response.ContentType = "text/plain";   //内容类型为纯文件
        context.Response.Write(DateTime.Now.ToString("T"));
                                                        //写入当前时间
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

（4）在项目中添加一个新 HTML 页面 AjaxTimePage.htm，页面代码如下：

```
<head>
    <title>XMLHttpRequest 对象示例</title>
    <script type="text/javascript" src="HttpFactory.js"></script>
     <script type="text/javascript">
        function test() {
            var request = HTTP.newRequest();    //创建 XMLHttpRequest
            request.onreadystatechange = function () {
                //检测 ajax 调用是否成功
                if (request.readyState == 4) {
                    if (request.status == 200)
                        //显示服务器返回的内容（即当前时间）
                        document.getElementById("time").innerHTML = request.
                        responseText;
                }
            };
             //创建和发送请求
            request.open("GET", "TimeHandler.ashx");
            request.send(null);
        }
    </script>
</head>
<body>
    <div>
    <input type="button" id="b1" onclick="test();" value="TEST" />
    当前服务器时间为：<span id="time"></span>
    </div>
</body>
</html>
```

（5）运行 AjaxTimePage.htm 页面，单击“刷新”按钮以获得服务器时间，可以看到，页面能够无刷新地获取并显示服务器返回的时间。

9.2　ASP.NET AJAX 基本控件

ASP.NET AJAX 框架主要包括 5 个核心控件，ScriptManager 控件、UpdatePanel 控件、UpdateProgress 控件、Timer 控件、ScriptManagerProxy 控件。通过这些控件可以方便地开发出基本的 AJAX 应用。

9.2.1　ScriptManager 控件

通过 AJAX 的全名和 9.1 节的例子都可以看出，JavaScript 脚本在 AJAX 中起着至关重要的作用。浏览器端通过 JavaScript 向服务器提交请求、获得响应并更新页面。在 AJAX 应用中，需要编写大量的 JavaScript 代码，ASP.NET AJAX 的 ScriptManager 控件是一个 JavaScript 脚本的管理工具，起着容纳、组织、管理 JavaScript 脚本的作用。

在第一个使用 ASP.NET AJAX 的页面上都必须有且只有一个 ScriptManager 控件，而且该控件必须出现在其他 AJAX 控件之前。在页面上添加 ScriptManager 控件的方法与其他控件相同，从 Visual Studio 工具箱的 AJAX Extensions 面板中找到 ScriptManager 控件然后拖动到页面上即可。

9.2.2　ScriptManagerProxy 控件

由于每个页面上只能包含一个 ScriptManager 控件，如果已经在母版页中添加了该控件，则在内容页中不能再次添加。如果需要在内容页中使用 ScriptManager 提供的功能，这时候需要用到 ScriptManagerProxy 控件。

当已在父元素中定义 ScriptManager 控件时，使嵌套组件（如内容页和用户控件）可以将脚本和服务引用添加到页中。该控件既可以直接位于页面中，也可以间接位于嵌套组件或父组件内部。使用 ScriptManagerProxy 控件，可在母版页或宿主页已包含 ScriptManager 控件的情况下，将脚本和服务添加到内容页和用户控件中。

ScriptManagerProxy 控件也是在 Visual Studio 工具箱的 AJAX Extensions 面板中，使用时直接将其拖动到页面上即可。

9.2.3　UpdatePanel 控件

使用 UpdatePanel 控件，无须编写 JavaScript 脚本就可以实现 AJAX 的异步局部刷新页面。UpdatePanel 控件通过指定页中无须刷新整个页面即可更新的区域发挥作用。此过程由 ScriptManager 服务器控件和客户端 PageRequestManager 类来协调。当启用部分页更新时，控件可以通过异步方式发布到服务器。

通过使用异步回发，可将页更新限制为包含在 UpdatePanel 控件中并标记为要更新的页区域。服务器仅将受影响元素的 HTML 标记发送到浏览器。在浏览器中，客户端 PageRequestManager 类执行文档对象模型（DOM）操作以将现有 HTML 替换为更新的标记。

UpdatePanel 控件是一个容器控件，本身没有任何可显示的元素，可以用来在页面上标记出可以独立刷新的区域。UpdatePanel 的使用比较简单，在页面上放置一个 UpdatePanel 控件，并将需要局部更新的控件放到 UpdatePanel 中就可以了。

【例 9-2】 UpdatePanel 基本使用。

本例演示 UpdatePanel 的基本使用。为了突出学习 UpdatePanel，减少其他元素干扰，此次仍以获得服务器时间为例。

（1）创建一个 ASP.NET Web 应用程序，并添加一个页面 SimpleUpdatePanel.aspx。

（2）在页面上放置一个 ScriptManager 控件，对应代码如下：

```
<asp:ScriptManager runat="server">
</asp:ScriptManager>
```

（3）在页面上 ScriptManager 后面添加一个 UpdatePanel 控件，对应代码如下：

```
<asp:UpdatePanel runat="server">
</asp:UpdatePanel>
```

（4）在页面设计视图中，从工具箱拖动一个 Label 和一个 Button 控件到 UpdatePanel 控件中，如图 9.1 所示，页面对应代码如下：

```
<asp:UpdatePanel runat="server">
    <ContentTemplate>
        当前时间是：<asp:Label ID="Label1" runat="server" Text="">
        </asp:Label><br />
        <asp:Button ID="Button1" runat="server" Text="获得当前时间" onclick=
        "Button1_Click" />
    </ContentTemplate>
</asp:UpdatePanel>
```

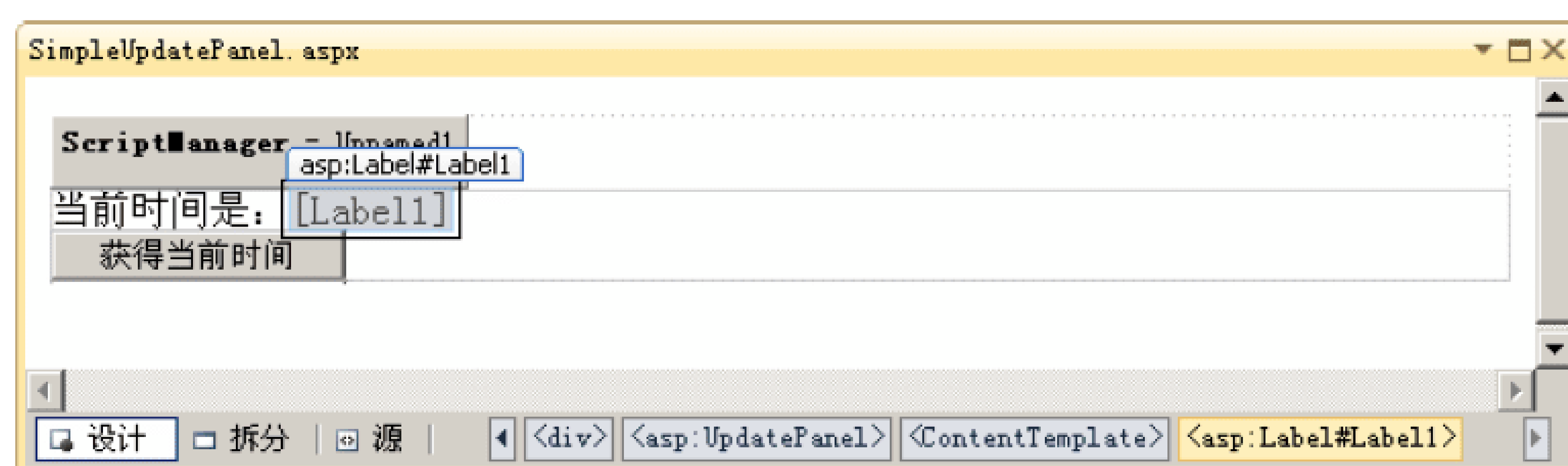


图 9.1　SimpleUpdatePanel 页面设计视图

（5）在“获得当前时间”按钮的 Click 事件中，编写以下代码。

```
protected void Button1_Click(object sender, EventArgs e)
{
    System.Threading.Thread.Sleep(500);
    Label1.Text = DateTime.Now.ToString("T");
}
```

（6）运行页面，单击“获得当前时间”按钮，可以看到，页面已经实现了 AJAX 效果。

例 9-2 是一个最简单的 UpdatePanel 使用例子，下面再看一个稍微复杂一点的例子。

【例 9-3】 使用 UpdatePanel 无刷新更新数据。

本例使用 UpdatePane 实现无刷新地加载 GridView 中的数据。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加一个从 Northwind 数据库生成的实体框架数据模型，其中包含 Products 和 Categories 表。

（3）在项目中添加一个页面 UpdatePanelPage.aspx，在页面上放置一个 ScriptManager 和一个 UpdatePanel 控件。

（4）在 UpdatePanelPage.aspx 页面的 UpdatePanel 控件中，添加一个 GridView 控件和一个 AspNetPager 分页控件，代码如下：

```
<form id="form1" runat="server">
<div>
    <asp:ScriptManager runat="server">
    </asp:ScriptManager>
    <asp:UpdatePanel runat="server">
        <%--UpdatePanel 控件的内容为一个 GridView 和一个 AspNetPager--%>
        <ContentTemplate>
        <asp:GridView runat="server" ID="grid"></asp:GridView>
        <webdiyer:AspNetPager ID="pager" runat="server" onpagechanged=
        "pager PageChanged"></webdiyer:AspNetPager>
        </ContentTemplate>
    </asp:UpdatePanel>
</div>
</form>
```

（5）在页面的 Page_Load 事件中，加载第一页数据。在分页控件的 PageChanged 事件中，加载当前面数据。

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
        loadData();
}
private void loadData()
{
    int size = pager.PageSize ;                          //页面大小
    int index = pager.CurrentPageIndex;                  //当前页码
    NorthwindEntities context = new NorthwindEntities(); //创建对象上下文
    var query = from p in context.Products
                select p;
    int count = query.Count();                           //得到商品总数
    pager.RecordCount = count;
    List<Product> list=query.OrderBy(p=>p.ProductID)
        .Skip(size * (index - 1))
        .Take(size).ToList();                            //取得当前页数据
    grid.DataSource = list;
    grid.DataBind();
}

protected void pager PageChanged(object sender, EventArgs e)
{
    loadData();
}
```

（6）运行页面，单击分页控件查看新的一页数据，可以看到能够实现无刷新地加载新的数据。运行界面如图 9.2 所示。

在前两个例子中，引起页面更新的控件和被更新的控件都在同一个 UpdatePanel 中，如例 9-2 中的 Button 和 Label 在同一个 UpdatePanel 中，例 9-3 中的 GridView 和 AspNetPager 控件也在同一个 UpdatePanel 中。如果引起页面更新的控件和被更新的控件不在同一个 UpdatePanel 中，则引起页面更新的事件发生时，就会变成一个普通的 ASP.NET 服务器端事件，产生整个页面同步回发。在这种情况下，如果仍然想获得 AJAX 效果，则需要使用 UpdatePanel 的触发器。UpdatePanel 的触发器是指能够触发 UpdatePanel 进行更新的事件。

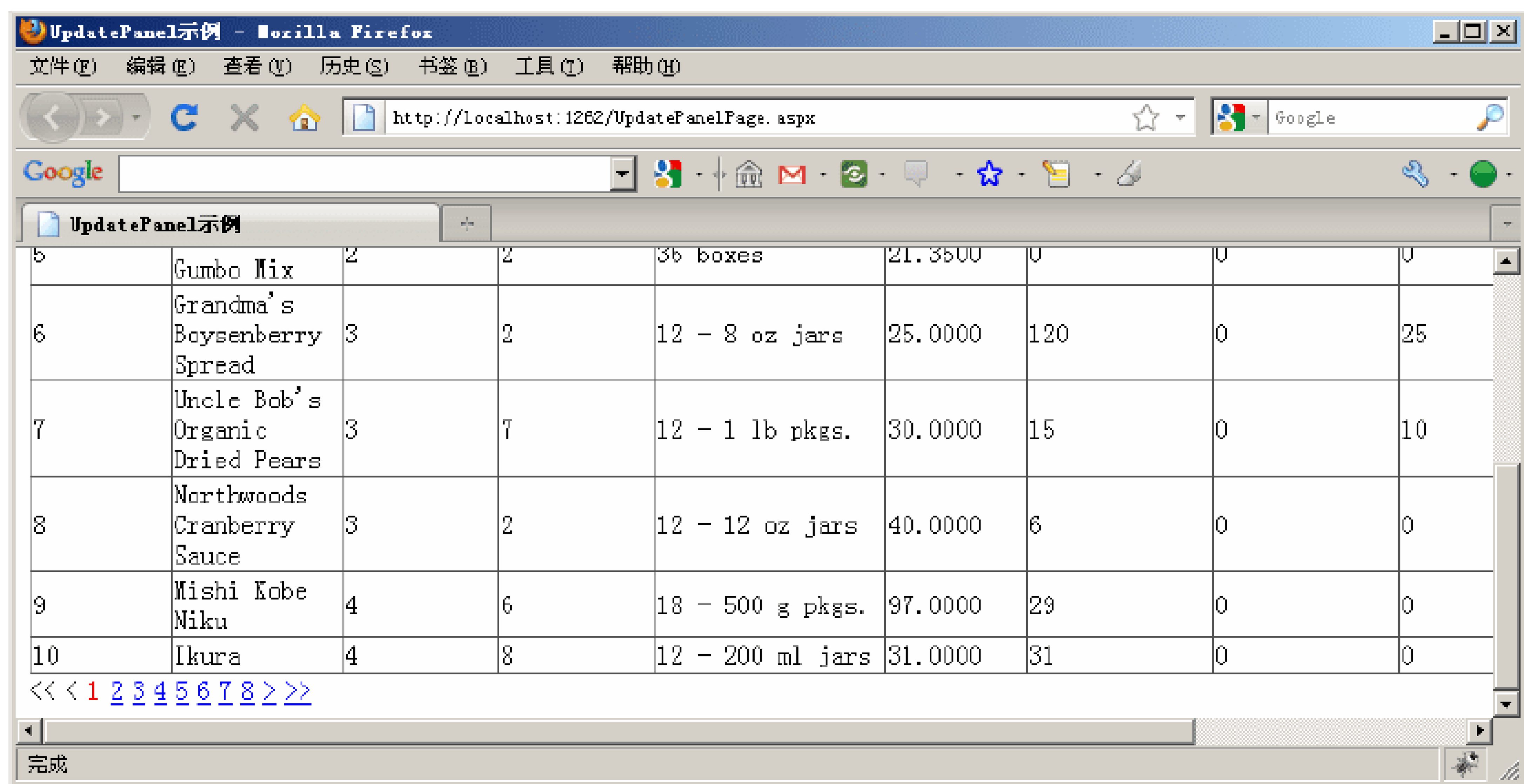


图 9.2　UpdatePanel 加载数据

【例 9-4】 UpdatePanel 触发器。

本例演示 UpdatePanel 控件触发器的使用。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加一个从 Northwind 数据库生成的实体框架数据模型，其中包含 Products 和 Categories 表。

（3）在项目中添加一个页面 TriggerPage.aspx，在页面上放置一个 ScriptManager 和一个 UpdatePanel 控件。

（4）在 UpdatePanel 控件中添加一个 GridView 控件。

（5）在 UpdatePanel 外添加一个 AspNetPager 控件，此时页面代码如下：

```
<div>
    <asp:ScriptManager runat="server">
    </asp:ScriptManager>
    <asp:UpdatePanel runat="server">
        <ContentTemplate>
        <asp:GridView runat="server" ID="grid"></asp:GridView>
        </ContentTemplate>
    </asp:UpdatePanel>
    <webdiyer:AspNetPager ID="pager" runat="server" onpagechanged="pager
    PageChanged">
    </webdiyer:AspNetPager>
</div>
```

（6）在页面设计视图中，选中 UpdatePanel 控件，从属性窗口中找到 Triggers 属性，单击右侧的省略号按钮，则弹出如图 9.3 所示的对话框。

（7）单击“添加”按钮右侧下拉箭头，从下拉菜单中选择 AsyncPostBackTrigger（异步回传触发器），然后从对话框右侧的 ControlID 下拉表中选择 pager 控件，从 EventName 中选择 PageChanged 事件，如图 9.4 所示。

（8）设置完毕触发器后，查看页面源码，包含以下代码。

```
<div>
    <asp:ScriptManager runat="server">
    </asp:ScriptManager>
    <asp:UpdatePanel runat="server">
        <ContentTemplate>
        <asp:GridView runat="server" ID="grid"></asp:GridView>
```

```
        </ContentTemplate>
        <%-- 定义触发器 --%>
        <Triggers>
        <asp:AsyncPostBackTrigger ControlID="pager" EventName="Page-
        Changed" />
        </Triggers>
    </asp:UpdatePanel>
    <webdiyer:AspNetPager ID="pager" runat="server" onpagechanged="pager_
    PageChanged"></webdiyer:AspNetPager>
</div>
</form>
```

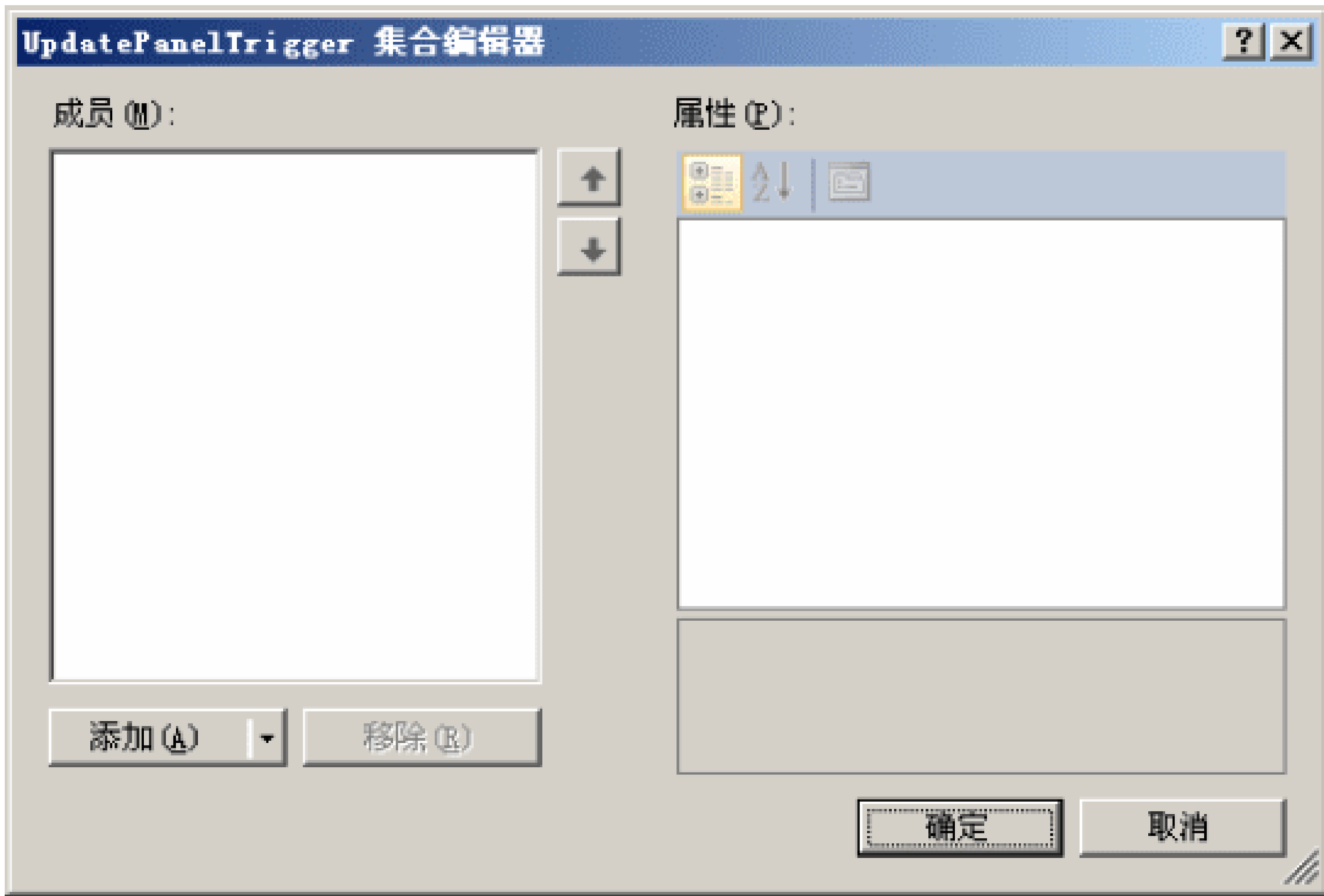


图 9.3　UpdatePanel Trigger 集合编辑器

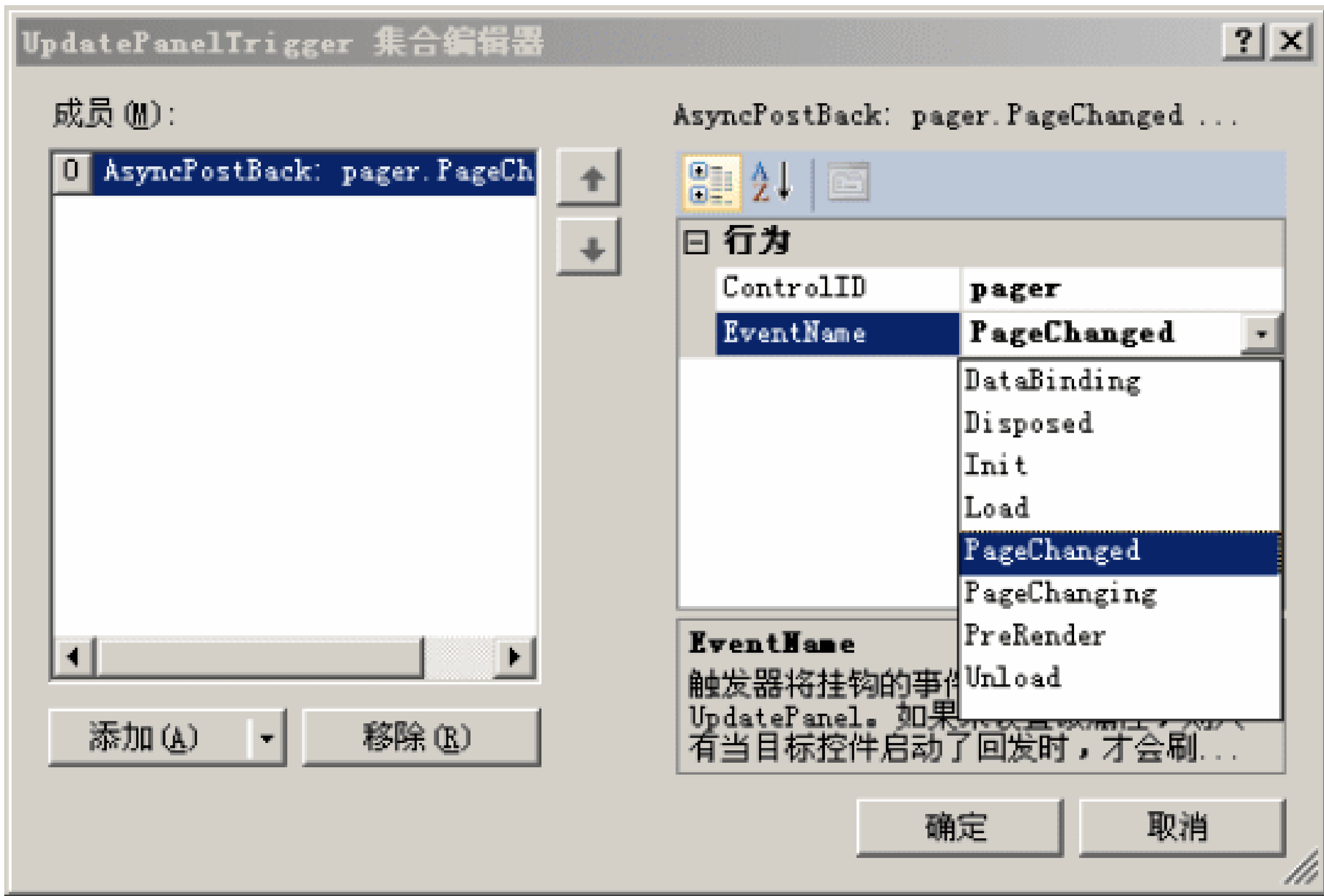


图 9.4　设置触发器属性

（9）运行 TriggerPage.aspx 页面，运行结果与例 9-2 相同。

9.2.4　UpdateProgress 控件

UpdateProgress 控件的作用类似于一个进度条，通常与 UpdatePanel 控件配合使用，当 UpdatePanel 正在更新数据时，UpdateProgress 控件中的内容（通常是更新动画或者提示文字）就会显示出来，当 UpdatePanel 更新完毕后，UpdateProgress 控件中的内容会自动隐藏。

【例 9-5】 UpdateProgress 控件示例。

本例通过 UpdateProgress 控件指示页面正在进行局部更新。

（1）打开例 9-3 或者例 9-2 创建的项目。

（2）在 UpdatePanel 前添加一个 UpdateProgress 控件，控件中写一段提示文字，放一幅加载动画，代码如下：

```
<div>
    <asp:ScriptManager runat="server">
    </asp:ScriptManager>
    <asp:UpdateProgress runat="server" ID="progress1">
        <%--UpdateProgress 控件模板--%>
        <progresstemplate>
         数据正在加载...<br />
         <img src="images/loading.gif" alt="正在加载..." style="height:
         100px;" />
         </progresstemplate>
    </asp:UpdateProgress>
    <asp:UpdatePanel runat="server">
        <ContentTemplate>
```

```
        <asp:GridView runat="server" ID="grid"></asp:GridView>
        <webdiyer:AspNetPager ID="pager" runat="server" onpagechanged=
        "pager_PageChanged"></webdiyer:AspNetPager>
        </ContentTemplate>
    </asp:UpdatePanel>
</div>
```

（3）为了使等待加载的动画效果更加明显，在 loadData()方法中添加一条语句，使线程休眠一段时间。

```
private void loadData()
{
    System.Threading.Thread.Sleep(1000);
    int size = pager.PageSize ;                              //页面大小
    int index = pager.CurrentPageIndex;                      //当前页码
    NorthwindEntities context = new NorthwindEntities();     //创建对象上下文
    var query = from p in context.Products
                select p;
    int count = query.Count();                               //得到商品总数
    pager.RecordCount = count;
    List<Product> list=query.OrderBy(p=>p.ProductID)
        .Skip(size * (index - 1))
        .Take(size).ToList();                                //取得当前页数据
    grid.DataSource = list;
    grid.DataBind();
}
```

（4）运行页面，运行结果如图 9.5 所示。

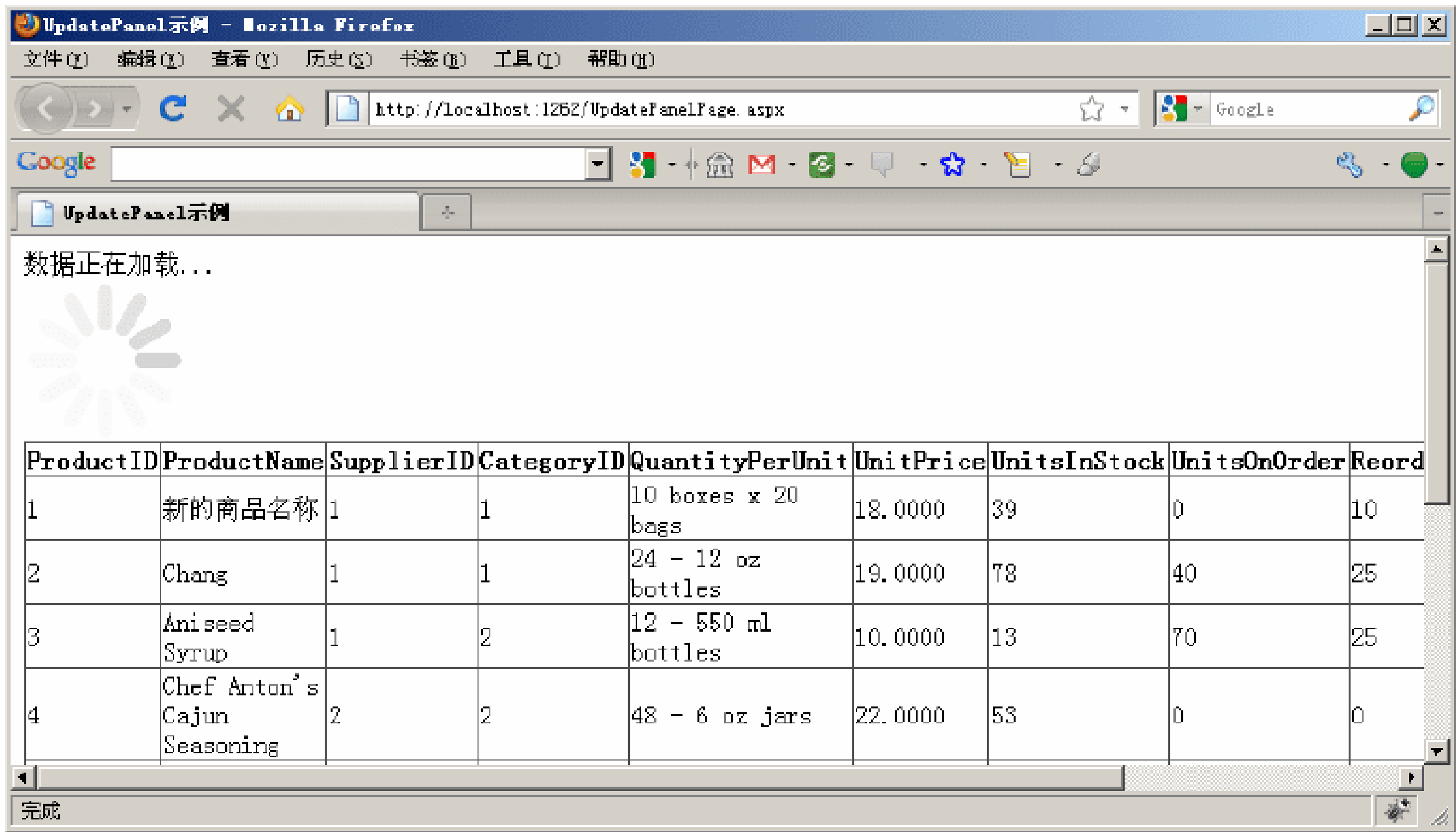


图 9.5　UpdateProgress 指示更新进度

9.2.5　Timer 控件

Timer 控件可以实现定时向服务器发送请求并触发服务器端事件，Timer 控件通常用于执行周期性的任务。

【例 9-6】 自动更新时间。

本例演示通过 Timer 控件周期性自动获取并显示服务器时间的功能。本例用到了 UpdatePanel 的 Trigger 属性。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加一个页面 TimerPage.aspx。

（3）在 TimerPage 页面上放置一个 ScriptManager 控件、一个 UpdatePanel 控件和一个 Timer 控件。在 UpdatePanel 控件中放置一个 Label 用以显示当前时间。在 UpdatePanel 外面放置一个 Timer 控件，将 Timer 控件的 Tick 事件添加到 UpdatePanel 的触发器中。代码如下：

```
<div>
    <asp:ScriptManager runat="server">
    </asp:ScriptManager>
    <asp:UpdatePanel runat="server">
        <%--UpatePanel 内容模板--%>
        <contenttemplate>
        服务器当前时间为：<asp:Label runat="server" id="label1" Text="Label">
        </asp:Label>
        </contenttemplate>
        <%--UpatePanel 控件的触发器--%>
        <triggers>
            <asp:AsyncPostBackTrigger ControlID="timer1" EventName=
            "Tick" />
        </triggers>
    </asp:UpdatePanel>
    <asp:Timer runat="server" id="timer1" ontick="timer1_Tick" Interval=
    "1000">
    </asp:Timer>
</div>
```

（4）在 Timer 控件的 Tick 事件中设置显示当前时间。

```
protected void timer1_Tick(object sender, EventArgs e)
{
    label1.Text = DateTime.Now.ToString("T");
}
```

（5）运行页面，可以看到，页面每隔 1 秒就会自动更新服务器时间。

9.3　ASP.NET AJAX 控件工具箱简介

为了方便用户开发功能更强大的 AJAX 应用程序，微软公司开发了一个 ASP.NET AJAX 控件工具箱（ASP.NET AJAX Control Toolkit），其中包含了一些常用的 AJAX 功能和控件。本节将介绍 ASP.NET AJAX 控件工具箱的用法及几个常用控件。

9.3.1　下载和安装

默认情况下 Visual Studio 2010 中并不包含 ASP.NET AJAX 控件工具箱，如果想使用的话，则需要到网站下载和安装。ASP.NET AJAX 控件工具箱官方网站和下载地址为：http://ajaxcontroltoolkit.codeplex.com/。下载后得到一个压缩包，解压后有一个 dll 文件和各

个国家的本地化资源文件，把这些文件复制到磁盘上的一定路径下，然后从 Visual Studio 中打开工具箱，右击工具箱空白处，从弹出的快捷菜单中选择“选择项”选项，则会打开如图 9.6 所示的对话框。

在图 9.6 所示的选择工具箱项对话框中单击“浏览”按钮，从弹出的新窗口中选择刚才解压以后得到的 dll 文件，然后单击“确定”按钮。此时可以从 Visual Studio 工具箱中看到新添加的 AJAX 工具了，如图 9.7 所示。

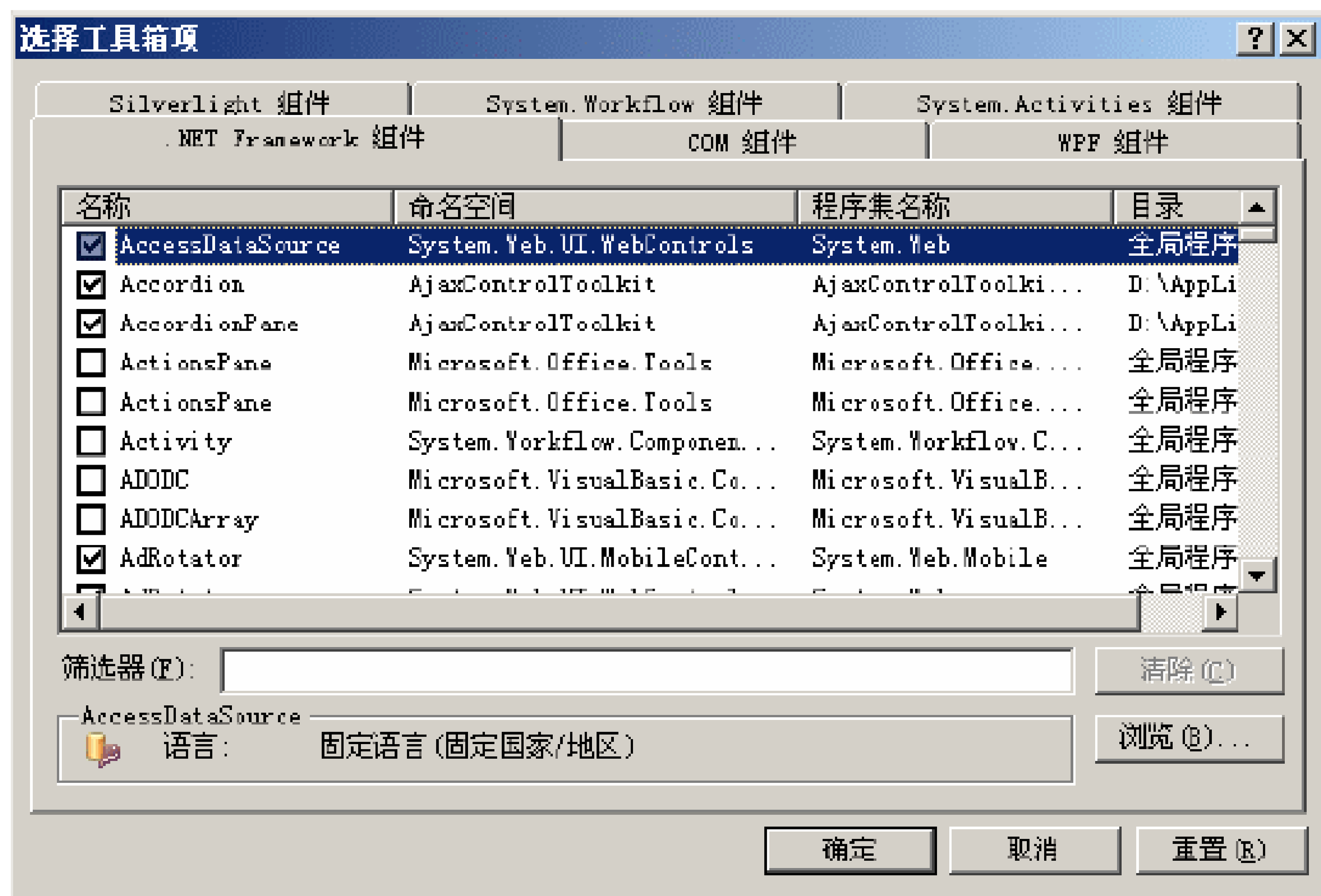


图 9.6 “选择工具箱项”对话框

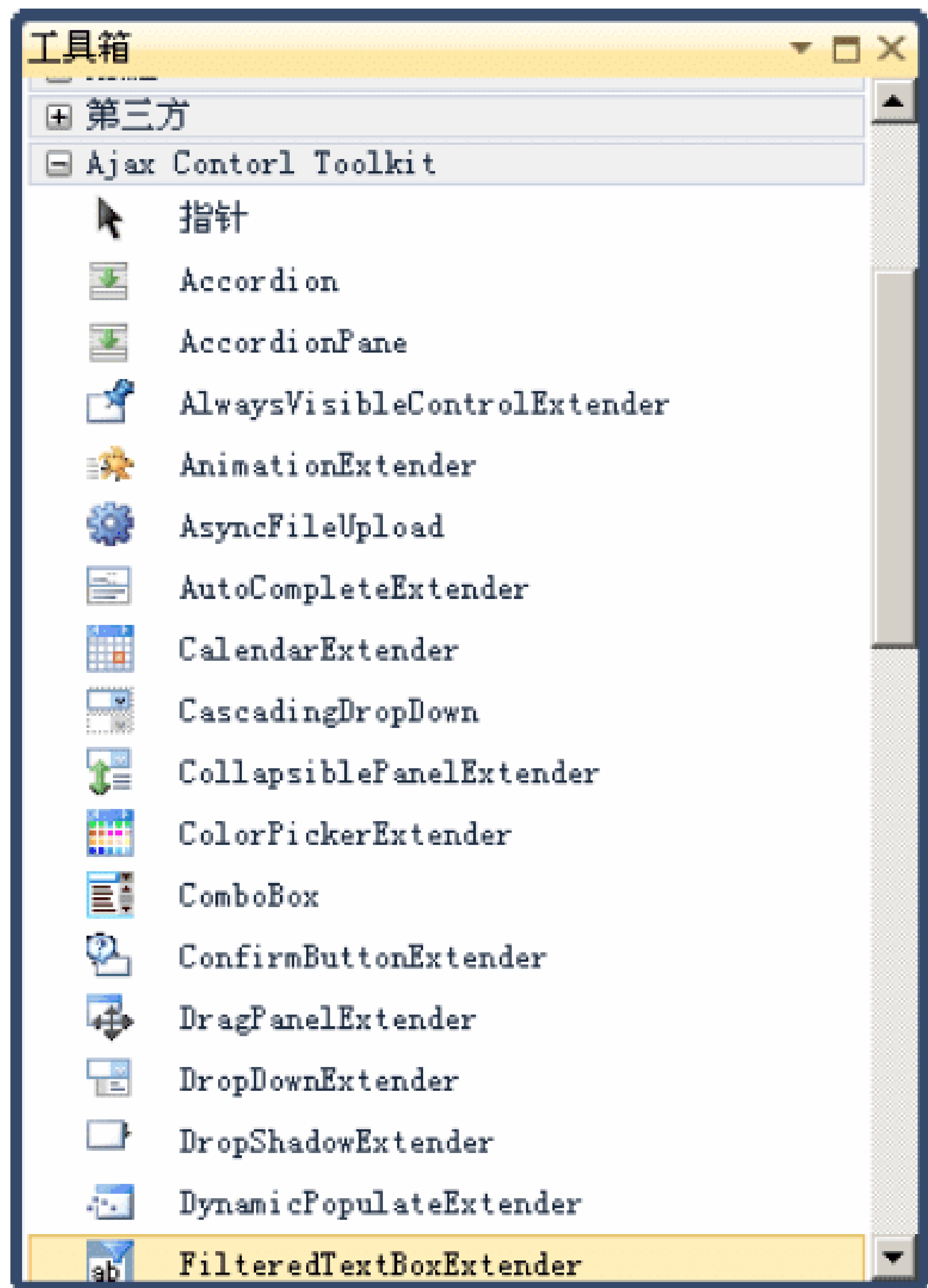


图 9.7　AJAX 控件工具箱

9.3.2　应用举例

ASP.NET AJAX Control Toolkit 中的组件大致可以分为两类：一类是独立的控件，这类控件就像普通控件一样，直接放到页面上使用。另外一类是扩展控件，这类控件本身并不是一个独立的控件，而是通过对其他现有控件（如 TextBox、Button 等）进行扩展来实现功能。ASP.NET AJAX Control Toolkit 中包含很多控件，限于篇幅，此处不能一一加以介绍。这里只分别讲解两类控件中的一种，其他控件的使用方法与此类似，具体细节可参考微软官方文档。

在使用基本 AJAX 控件时，页面上需要放置一个 ScriptManager 控件。与此类似，在使用 AJAX Control Toolkit 控件里，也需要在页面上放置一个脚本管理控件 ToolkitScriptManager。

【例 9-7】 Accordion 控件。

Accordion 控件称为可折叠面板控件，可以实现像 QQ 好友列表那种分组折叠的效果。该控件属于 ASP.NET AJAX Control Toolkit 中的独立控件一类。本例将演示该控件的使用方法。

（1）创建一个 ASP.NET Web 应用程序，在项目中添加一个页面 AccordionPage.aspx。

（2）在页面上添加一个 ToolkitScriptManagerToolkitScriptManager 控件，对应代码如下：

```
<asp:ToolkitScriptManager ID="ToolkitScriptManager1" runat="server">
</asp:ToolkitScriptManager>
```

（3）从工具箱中拖动一个 Accordion 控件到页面上，对应代码如下：

```
<asp:Accordion ID="Accordion1" runat="server" >
</asp:Accordion>
```

（4）为了使 Accordion 控件具有更好的外观效果，需要为其指定多个 CSS 样式，包括面板标题样式 HeaderCssClass、面板内容样式 ContentCssClass、选中面板的标题样式 HeaderSelectedCssClass，代码如下：

```
<asp:Accordion              ID="Accordion1"              runat="server"
HeaderCssClass="accordionHeader"      ContentCssClass="accordionContent"
HeaderSelectedCssClass="accordionHeaderSelected" >
</asp:Accordion>
```

（5）在 Accordion 控件中添加多个面板。本例添加了 3 个面板，分别是基本信息、编程技术和工作经验，代码如下：

```
<asp:Accordion ID="Accordion1" runat="server"  HeaderCssClass="accordion-
Header" ContentCssClass="accordionContent" HeaderSelectedCssClass="acco-
rdionHeaderSelected" >
<Panes>
    <asp:AccordionPane ID="AccordionPane1" runat="server">
    <Header>基本信息</Header>
    </asp:AccordionPane>
     <asp:AccordionPane ID="AccordionPane2" runat="server">
    <Header>编程技术</Header>
    </asp:AccordionPane>
     <asp:AccordionPane ID="AccordionPane3" runat="server">
    <Header>工作经验</Header>
    </asp:AccordionPane>
</Panes>
</asp:Accordion>
```

（6）分别为 3 个面板添加内容，代码如下：

```
<asp:Accordion ID="Accordion1" runat="server"  HeaderCssClass="accordion-
Header" ContentCssClass="accordionContent" HeaderSelectedCssClass="acco-
rdionHeaderSelected" >
<Panes>
    <asp:AccordionPane ID="AccordionPane1" runat="server">
    <Header>基本信息</Header>
    <Content>
    姓名：<asp:TextBox ID="TetxtBox1" runat="server" /><br />
    年龄：<asp:TextBox ID="TextBox2" runat="server" /><br />
    生日：<asp:TextBox ID="TextBox3" runat="server" /><br />
    性别：<asp:RadioButton Text="男" ID="male" runat="server" />
    <asp:RadioButton Text="女" ID="female" runat="server" /><br/>
     学历：<asp:TextBox ID="TextBox4" runat="server" />
    </Content>
    </asp:AccordionPane>
     <asp:AccordionPane ID="AccordionPane2" runat="server">
```

```
    <Header>编程技术</Header>
    <Content>
    请在这里详细填写你所掌握的编程技术：<br />
        <asp:TextBox runat="server" TextMode="MultiLine" Height="200"
        Width="400" />
    </Content>
    </asp:AccordionPane>
     <asp:AccordionPane ID="AccordionPane3" runat="server">
    <Header>工作经验</Header>
    <Content>
    请在这里详细填写你的工作经验和项目经验：<br />
        <asp:TextBox ID="TextBox5" runat="server" TextMode="MultiLine"
        Height="200" Width="400" />
    </Content>
    </asp:AccordionPane>
</Panes>
</asp:Accordion>
```

（7）运行 AccordionPage.aspx 页面，运行结果如图 9.8 所示。

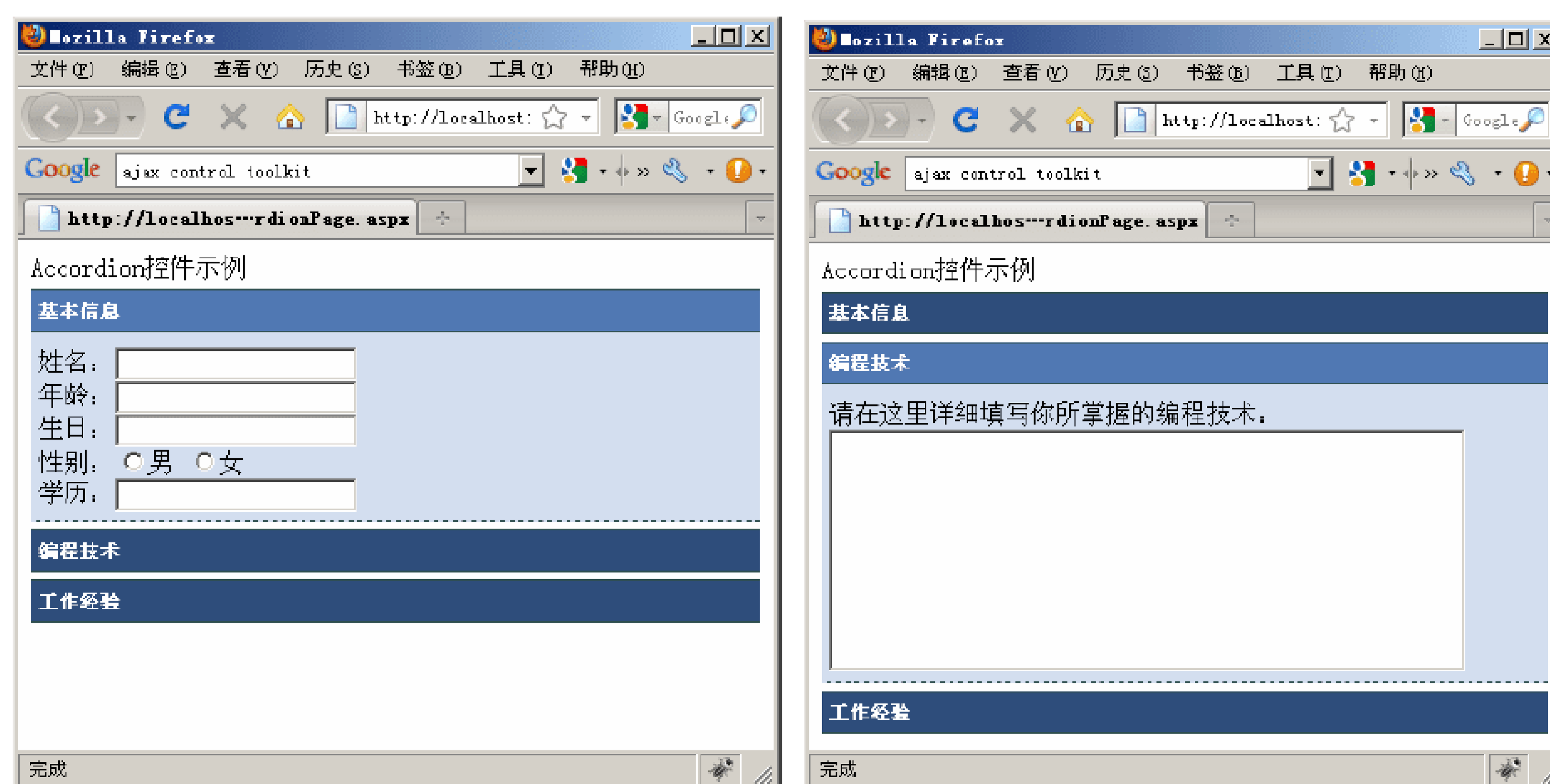


图 9.8　Accordion 控件示例

【例 9-8】 Calender 扩展控件。

ASP.NET AJAX Control Toolkit 中的日历控件是一种扩展控件，可以对 TextBox 进行扩展使其变成一个日历控件。本例演示 Calendar 扩展控件的使用。

（1）创建一个 ASP.NET Web 应用程序，添加一个页面 AjaxCalendarPage.aspx。

（2）在页面上添加一个 ToolkitScriptManagerToolkitScriptManager 控件。

（3）在 AjaxCalendarPage.aspx 页面上放置一个 TextBox 控件。在设计视图中，选中这个 TextBox 控件，则在控件右上角会出现一个箭头按钮，单击这个按钮，会弹出“TextBox 任务”面板，上面有一个“添加扩展程序”选项，如图 9.9 所示。

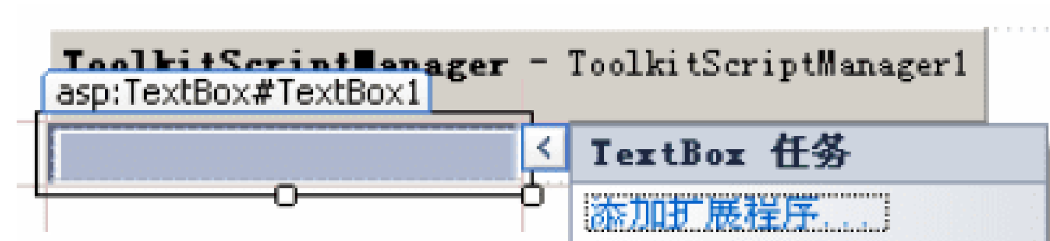


图 9.9　为 TextBox 添加扩展程序

🔔提示：如果 Visual Studio 2010 开发环境中没有出现图 9.9 所示的“添加扩展程序”界面，则说明尚未安装 ASP.NET AJAX Control Toolkit，按照本节开始所讲述的方法，将 ASP.NET AJAX Control Toolkit 添加到 Visual Studio 2010 开发环境的工具箱中，就会出现这个界面了。

（4）选择图 9.9 所示的“添加扩展程序”选项，则弹出“扩展程序向导”对话框，其中显示了可添加到所选择的 TextBox 控件的所有扩展程序，如图 9.10 所示。从中选择 CalendarExtender，并单击“确定”按钮。

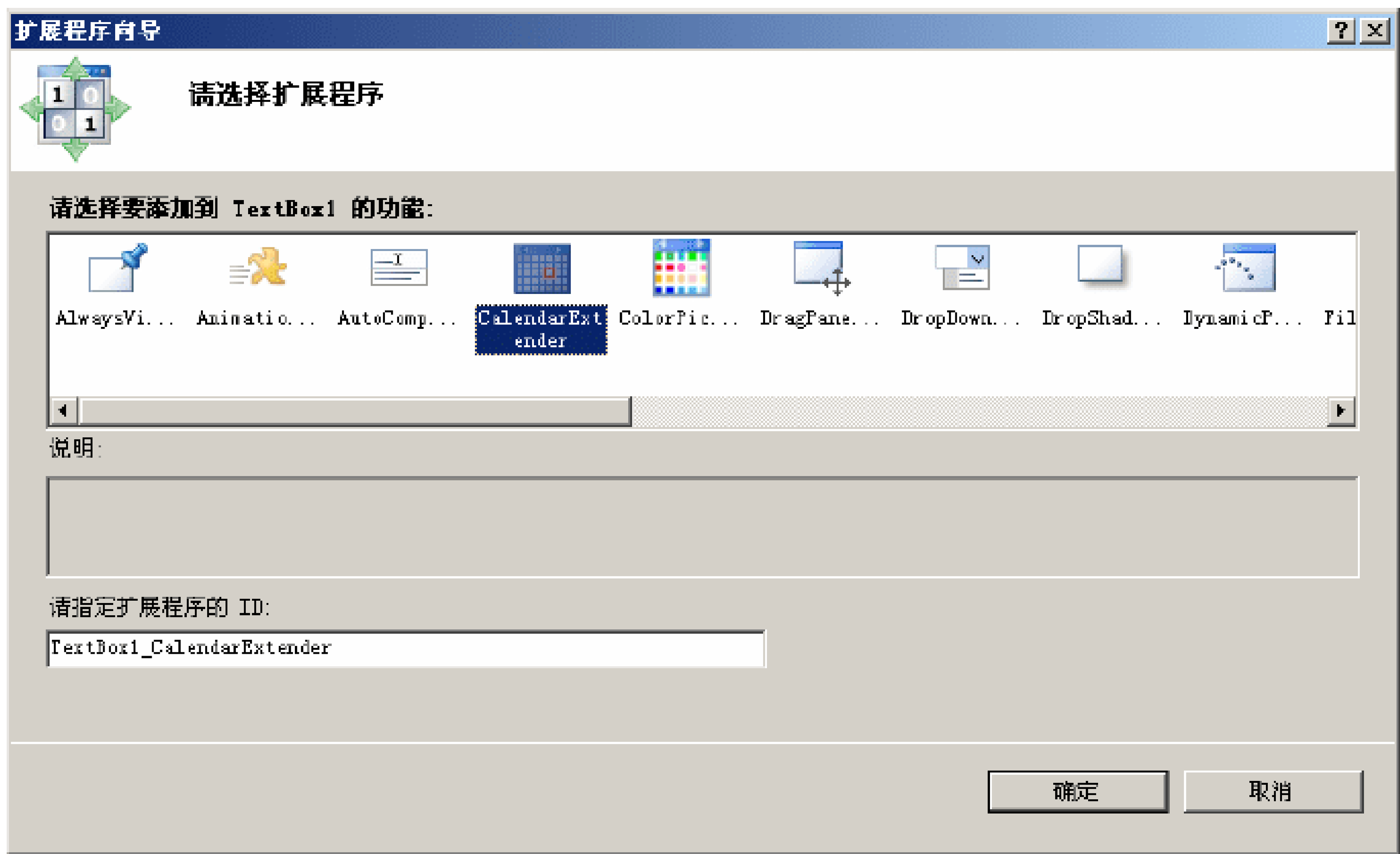


图 9.10　扩展程序向导

完成这一步操作后，AjaxCalendarPage.aspx 页面代码如下：

```
<asp:ToolkitScriptManager ID="ToolkitScriptManager1" runat="server">
</asp:ToolkitScriptManager>
<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
<asp:CalendarExtender ID="TextBox1 CalendarExtender" runat="server"
    Enabled="True" TargetControlID="TextBox1">
</asp:CalendarExtender>
```

（5）为 TextBox 添加了 Calender 扩展后，运行 AjaxCalendarPage.aspx 页面，在浏览器中单击 TextBox 控件，可以看到，会自动弹出一个日历控件。在日历控件中选择一个日期，所选中的日期就会自动设置到 TextBox 中。运行界面如图 9.11 所示。

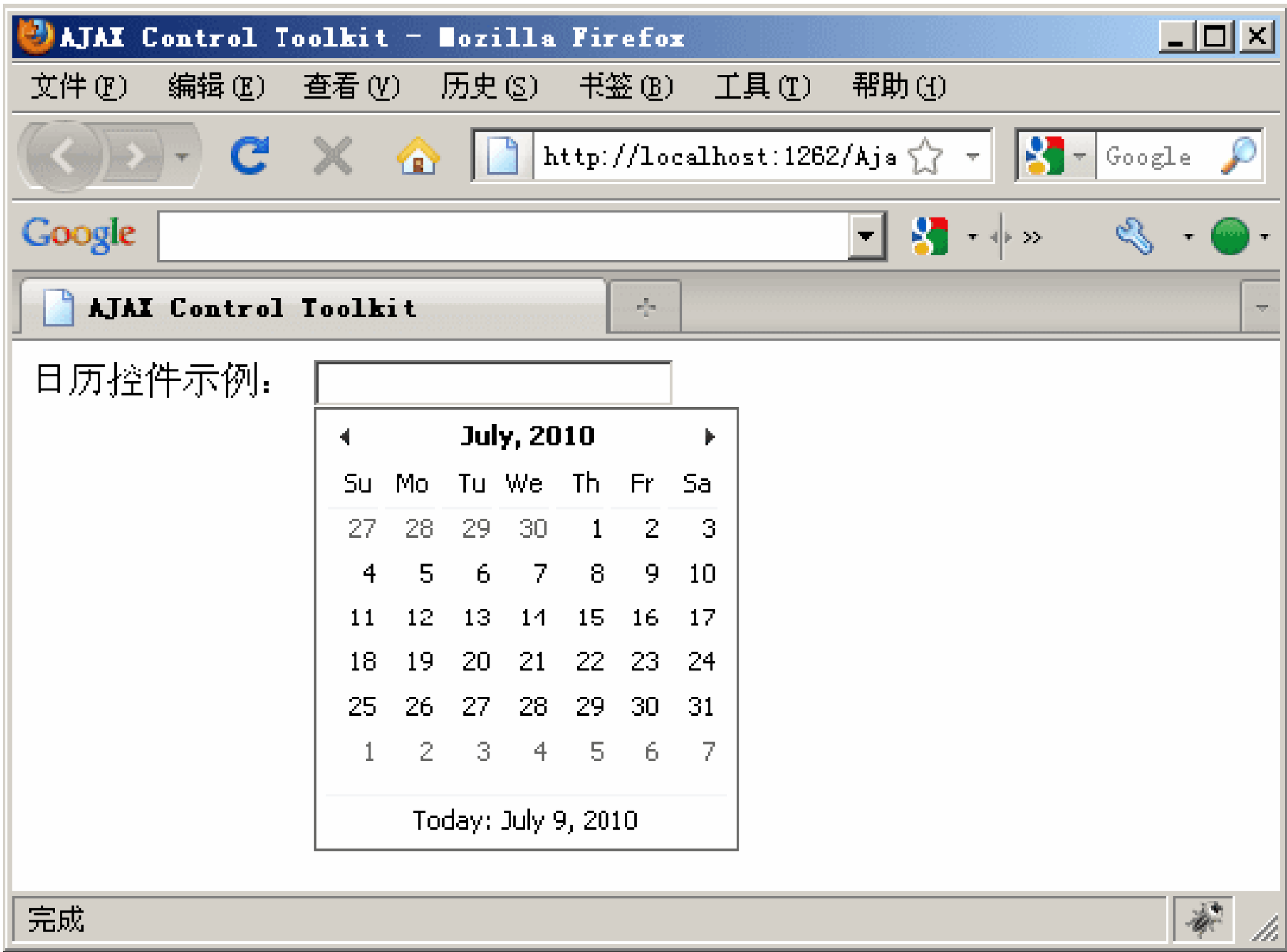


图 9.11　日期扩展控件示例

9.4　小　　结

本章介绍了 AJAX 开发相关技术，包括 AJAX 原理、ASP.NET AJAX 基本控件和 ASP.NET AJAX Control Toolkit。其中对 ASP.NET AJAX Control Toolkit 只进行了简单的介绍，说明了 Accordion 控件和 CalendarExtender 控件的简单应用。更多的控件及更详细的使用方法，读者可以参见微软官方网站。

第 10 章　优秀的 JavaScrip 框架 jQuery

随着 AJAX 技术的广泛应用和胖客户端 Web 程序的逐渐流行，Web 开发人员需要编写越来越多的 JavaScript 代码，与服务器后台进行异步交互，操作浏览器页面上的元素。使用原始的 JavaScript 代码完成这些工作，需要编写大量重复代码。为了提高 JavaScript 编码效率，软件开发领域逐渐出现了一些 JavaScript 框架，如 jQuery、ExtJs、Prototype 等。其中 jQuery 是最优秀的和应用最广泛的 JavaScript 框架之一，而且 jQuery 受到 Visual Studio IDE 的支持，能够显示智能提示。对于 ASP.NET 开发人员来说，选择 jQuery 作为 JavaScript 和 AJAX 框架是一个很好的选择。

10.1　jQuery 简介

jQuery 是一个优秀的 JavaScript 框架，语法简洁直观，功能强大。本节将对 jQuery 的基础知识进行介绍，为深入学习打好基础。

10.1.1　为什么使用 jQuery

JavaScript 通常用来实现一些动态网页特效，例如当鼠标划过某商品时弹出一个操作菜单，菜单上面有“加入收藏”、“购买”等选项。JavaScript 还经常用于和后台服务器交互并显示得到的结果，例如在省、市、县 3 个下拉列表之间建立级联，选择一个省，则动态地加载该省所有市，选择一个市，则动态加载该市的所有县。类似这种工作，如果用原始的 JavaScript 来写，代码量很大，而且重复代码很多。使用 jQuery 完成同样的功能可以大大减少代码量。

在页面中编写 JavaScript 时，比较提倡的做法是将 JavaScript 脚本与 HTML 页面元素完全分离。但是在传统的 JavaScript 编写中，经常需要将 JavaScript 函数混在 HTML 标记中。例如，如果要在单击某按钮时执行一个 JavaScript 函数，则通常是在 button 标记的 onclick 属性中指定函数名称，如下代码所示。

```
<input type="button" value="测试按钮" onclick="fun1()" />
```

这种方式将 JavaScript 代码与 HTML 元素混在一起，不易维护。而使用 jQuery 时，则不需要在 HTML 元素中通过 onclick 属性指定函数名称，从而将 JavaScript 脚本与 HTML 元素彻底分离。

jQuery 中还封装了强大的 AJAX 功能。通过第 9 章的例子可以看到，如果使用原始的

JavaScript 脚本实现 AJAX 功能，编码较为复杂。使用 jQuery 可以将工作大大简化。

另外，jQuery 是一个开放的框架，很容易对 jQuery 进行扩展。目前网络上已经有很多各种功能的 jQuery 插件，对于大多数功能来说，开发人员都能够找到相应的 jQuery 插件。

总之，jQuery 是一个很优秀的 JavaScript 框架，封装了很多常用功能，而且使用简单，可读性强，其语法甚至比原始的 JavaScript 还要容易掌握。通过 jQuery，开发人员只需要编写少量代码就可以完成复杂的工作，这也正是 jQuery 宣传的口号：写得更少，做得更多（Write Less, Do More）。

10.1.2　下载和使用 jQuery

jQuery 是作为一个 js 文件来分布的，可以从 jQuery 官方网站 http://jquery.com/下载到最新的 jQuery。jQuery 有几个版本，作为 ASP.NET 开发人员，可能用到 3 个版本；即压缩版、开发版和 Visual Studio 文档版。其中开发版是标准的 JavaScript 文件，具有注释和空格、换行。压缩版是将标准版中的所有空白和注释删除以后得到的版本，这个版本文件更小，下载速度更快，能够提高网页加载速度。

Visual Studio 文档版是专门为 Visual Studio 集成开发环境做的版本，使用此版本可以在 Visual Studio 中显示 jQuery 的智能提示，如图 10.1 所示。

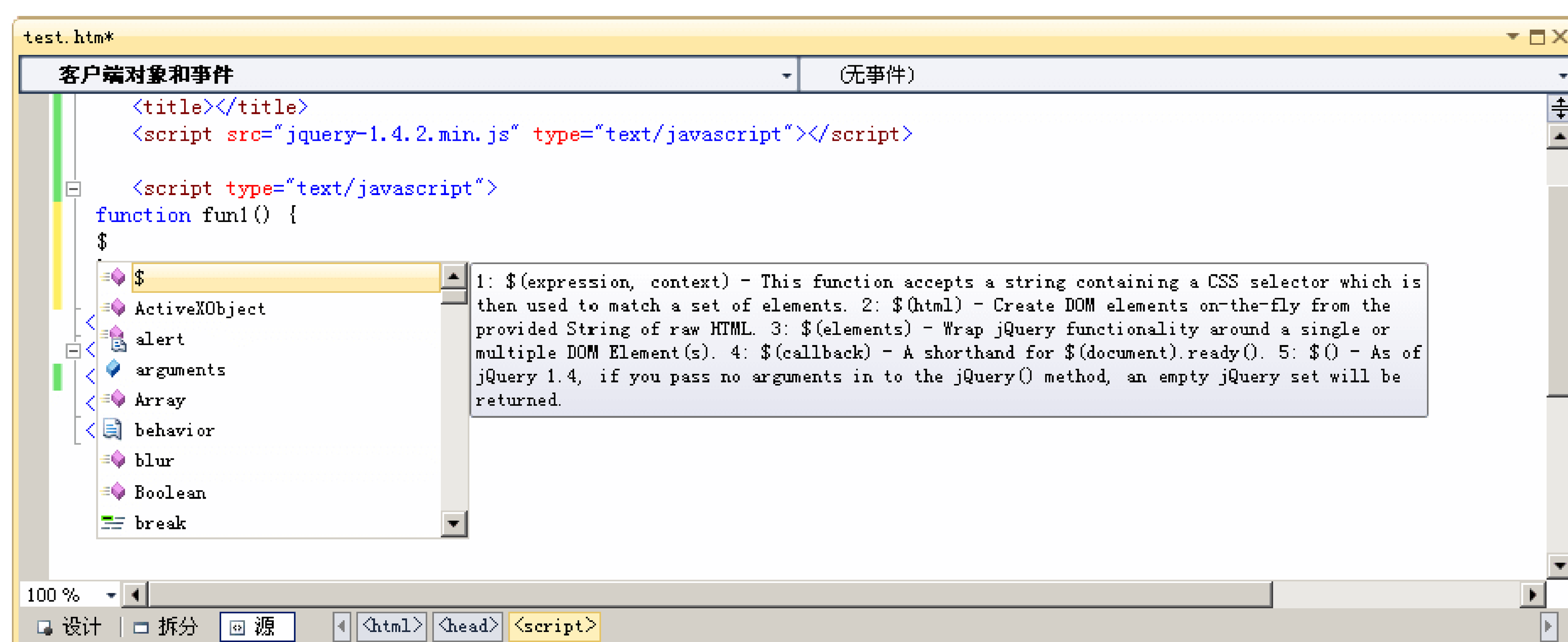


图 10.1　Visual Studio 对 jQuery 语法的智能提示

根据 3 个版本的特点，一般在开发时使用 jQuery 的压缩版和 Visual Studio 文档版，阅读 jQuery 代码时可以使用开发版，发布程序时使用压缩版。在页面中引用 jQuery 的方法与引用其他 JavaScript 文件相同，使用 script 标记通过 src 属性指定要加载的文件，如下代码所示。

```
<script src="jquery-1.4.2.min.js" type="text/javascript"></script>
```

10.1.3　jQuery 和$

jQuery 中的$是最经常使用的一个函数。其实 jQuery 和$是完全特价的，$就是 jQuery 的别名，只不过使用$语法更简洁，语句更短。从以下 jQuery 的源码中可以明确地看出$和 jQuery 是同一个函数。

```
window.jQuery = window.$ = jQuery;
```

也可以自己写一段代码来验证$与 jQuery 相等，代码如下：

```
function fun1() {
    if ($ === jQuery)
        alert('$与 jQuery 完全相等。');                //运行时将显示这个提示
    else
        alert('$与 jQuery 不是同一个对象。');
}
```

根据参数的不同，$函数主要有 5 种用法，实现 5 种不同的作用。

（1）首先介绍$函数的第 1 种用法，其作用为文档加载完成时执行一个函数，此时$函数参数是一个函数（匿名函数或者命名函数），其语法有以下两种：

```
$(function() { /*在这里写函数体*/ });            //参数为一个匿名函数
$(myFunction);                                   //其中 myFunction 是一个函数
```

下面举例说明$函数的这种用法，创建一个 HTML 页面并在其中写如下代码：

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title> jQuery $函数示例</title>
   <script src="jquery-1.4.2.min.js" type="text/javascript"></script>
   <script type="text/javascript">
      $(function () {
         alert('当 HTML 文档（document）加载完成时，就会执行这段代码。');
      });
   </script>
</head>
<body>jQuery $函数示例</body>
</html>
```

运行上述 HTML 页面，则页面加载后立即弹出消息提示。

（2）$函数的第 2 种用法是选择 HTML 页面上一组元素，并将其作用 jQuery 集合返回。其语法为：

```
$(expression, context)
```

其中第 1 个参数 expression 是一个字符串参数，表示查找元素的条件，这个参数通常称为选择器（selector）。这个术语来源于 CSS，在 CSS 中，如果要设置某些元素的样式，则首先要找到这些元素，就是通过选择器来指定需要应用样式的元素。jQuery 使用与 CSS 基本相同的选择器选择 DOM 元素。第 2 个参数指定了查找的范围，$函数将在该范围中查找选择器所指定的元素。如果不指定第 2 个参数，则默认值为 document，即在整个 HTML 文档范围内查找。看一个这种用法的例子，下面的代码将页面上所有 div 背景色设置为浅绿色。

```
$('div').css("background-color", "#eeffee");
```

（3）$函数的第 3 种用法是根据字符串生成相应的 DOM 元素，其语法为：

```
$(html)                          //html 为一个字符串，表示要创建的 DOM 元素
```

例如，下面的代码创建了一个 div 元素，其中包含一个 p 元素。

```
$('<div style="border:dashed 1px silver;"><p>这是一段文字</p> </div>')
```

（4）$函数的第 4 种用法是将一个或多个 DOM 元素封装为 jQuery 集合，语法如下：

```
$(elements)                         //elements 为一个或多个 DOM 元素
```

（5）$函数的第 5 种用法是返回一个空的 jQuery 集合，这种用法没有参数。

10.2　操作 DOM 元素

使用 JavaScript 或者 jQuery 最常见的功能就是操作 DOM 元素，包括读取设置其文本，修改其样式，添加、删除子元素等。本节将介绍如何通过 jQuery 实现常见的 DOM 操作。

10.2.1　处理事件

按照软件工程规范，应该把所有 JavaScript 代码与 HTML 元素分离。在使用传统的 JavaScript 处理事件时，通过设置 DOM 元素属性的方式指定其事件处理程序，这样就不可避免地将一部分代码（至少是函数名）写进了 HTML 元素中。使用 jQuery 可以很方便地完全将代码与 HTML 元素相分离，将 JavaScript 代码单独写在<head>标记中或者 js 文件中。使用 jQuery 为 DOM 元素添加事件处理程序语法如下：

```
$(选择器).事件(函数)
```

上述代码表示当选择器所选中的元素发生指定事件时，将执行指定函数。例如，下面的代码表示单击 button1 时显示一个提示信息。

```
$('#button1').click(
    function ()
    { 'hello, jQuery.' }
);
```

10.2.2　处理元素内容

对于 DOM 元素的一个常见操作就是读取或者设置其内容，例如，获取 TextBox 的值，设置 div 标记中的内容，获取或设置表格某单元格内文本等。jQuery 主要提供两个函数（html()和 val()）来实现这些功能。

第 1 个函数是 html()函数，该函数有两个作用，如果函数没有参数，则获取元素内的 HTML 内容；如果函数有一个字符串参数，则将元素内容设置为字符串参数所表示 HTML 元素。其语法如下：

```
html()                  //返回元素内部的 HTML 内容
html(content)           //content 为字符串，将元素内容设置为 content 参数所指定值
```

【例 10-1】　使用 html()函数处理元素内容。

本例演示 html()函数读取和设置元素内容。页面上有两个 div 和两个按钮，单击第 1

个按钮可以实现读取第 1 个 div 内容并以消息框的形式显示出来，单击第 2 个按钮可以将第 1 个 div 的内容复制到第 2 个 div 中。页面运行结果如图 10.2 所示。

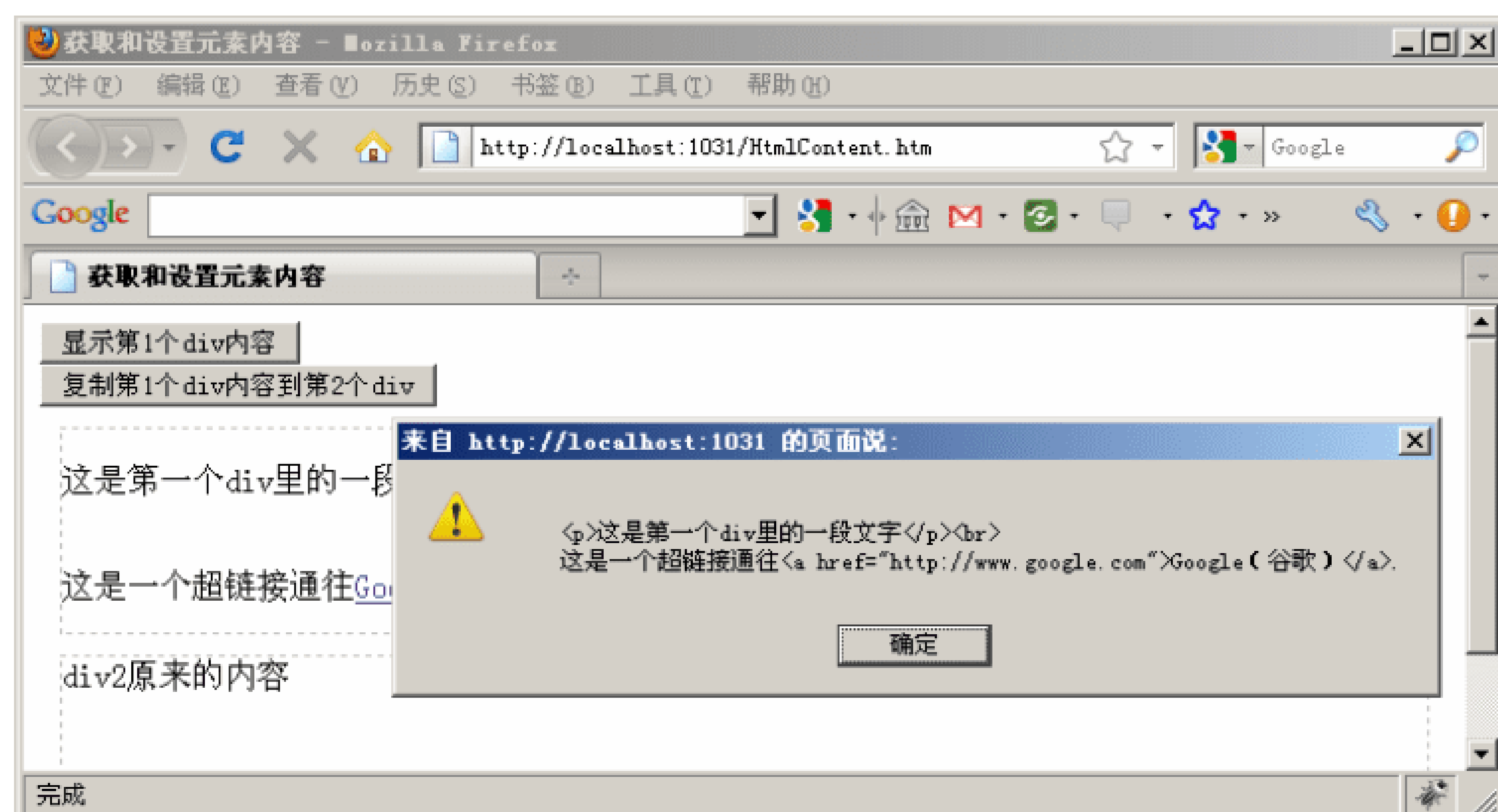


图 10.2　使用 html()函数处理元素内容

（1）在 Visual Studio 中创建一个 HTML 页面，在页面上放置两个 Button 和两个 div，代码如下：

```
<body>
<div id="main">
<input type="button" value="显示第 1 个 div 内容" id="showButton" /><br />
<input type="button" value="复制第 1 个 div 内容到第 2 个 div" id="copyButton" />
    <div id="div1">
    <p>这是第 1 个 div 里的一段文字</p><br />
    这是一个超链接通往<a href="http://www.google.com">Google（谷歌）</a>.
    </div>
    <div id="div2">
    div2 原来的内容
    </div>
</div>
</body>
```

（2）为了使两个 div 布局更加明显直观，为其添加 CSS 样式，设置边框和边距。

```
<style type="text/css">
/*设置 div 边框为灰色虚线，设计 div 间距，使两个 div 更容易区分*/
div#main > div
{ border:1px dashed silver; margin:10px; height:100px; }
</style>
```

（3）导入 jQuery 文件。

```
<script src="jquery-1.4.2.min.js" type="text/javascript"></script>
```

（4）编写 jQuery 代码，为两个按钮的 click 事件编写代码，分别实现读取和复制 div 内容的功能。

```
<script type="text/javascript">
    $(
    function () {
```

```
        $('#showButton').click(         //showButton 的 click 事件处理程序
        function () {
            var content = $('#div1').html();    //得到 div1 的内容
            alert(content);
        });
        $('#copyButton').click(            //showButton的click事件处理程序
          function () {
              var content = $('#div1').html(); //得到 div1 的内容
              $('#div2').html(content);        //将 div1 内容复制到 div2
          }
        );
    }                                          //最外层 function 结束
    );                                         //$结束
</script>
```

对于 jQuery 初学者来说，上述代码看起来可能有点复杂。$函数的参数是一个函数，这个函数有两个语句，其中每条语句又是一个函数（click 语句），这两个 click()函数的参数又分别是一个函数，函数中包含多条语句。像这种函数多层嵌套的语法静态语言（如 C#）中并不常见。为了使初学者更加清晰地理解上述代码，可以将其修改为以下功能等价的代码。

```
<script type="text/javascript">
    $( init );
    function init(){
        $('#showButton').click(click1);        //showButton 的 click 事件
        $('#copyButton').click(click2);        //copyButton 的 click 事件
    }
    function click1() {
        var content = $('#div1').html();       //得到 div1 的内容
        alert(content);
    }
    function click2() {
        var content = $('#div1').html();       //得到 div1 的内容
        $('#div2').html(content);              //将 div1 内容复制到 div2
    }
</script>
```

提示：上述两段代码功能完全相同，所不同的是，第 2 段代码在页面上添加了 3 个全局函数，而第 1 段代码没有添加任何命名函数。第 2 段代码由于添加了全局命名函数，就有可能造成与其他 js 文件中的函数命名冲突，通常称为污染命名空间。在编写 JavaScript 代码时，要尽可能少地添加全局成员（变量和函数等），以避免污染命名空间。在本章以后的示例中，将尽量使用第 1 种方式即匿名函数的方式写代码。

jQuery 第 2 个常用的操作元素内容函数为 val()函数。这个函数也有两种用法，如果函数没有参数，则获取元素的值；如果函数有参数，则将元素的值设置为参数的值。val()函数通常用来获取和设置 text、button 等 input 元素的值。例如，假设用户在一个 ID 为 yourName 的文本框中输入姓名，则可以使用以下语句向用户问候。

```
var name = $('#yourName').val();
alert('hello,' + name);
```

提示：并非所有 HTML 元素都支持 val()，只有在标记中包含 value 属性中元素（如 input type=text、input type=button 等）才能使用 val()读取或设置其值。对于不支持 value 属性的元素（如 div 等）使用 val()函数是没有意义的。

10.2.3　更改元素样式

jQuery 提供了多种途径更改元素的样式。其中一种最简单、最直观的方式是使用 css()函数。Css()函数可以设置或获取元素的 css 属性，语法如下：

```
css(name)                                    //获取名称为 name 的 css 属性值
css(name, value)                             //将名为 name 的 css 属性设置为 value
```

下面的代码是使用 css()函数的例子。

```
var back=$('#div1').css("background-color");        //获取 div1 的背景色
 $('#div2').css("background-color","#eeffee");      //设置 div2 背景色为浅绿色
```

使用 jQuery 设置元素样式的第 2 种方法为使用 addClass()函数和 removeClass()函数。addClass()向元素添加 CSS 类，removeClass()函数从元素删除 CSS 类。所添加的 CSS 类应该在页面中定义。下面的例子演示 addClass()和 removeClass()函数的使用。

```
$('#div1').addClass('green');                       //向 div1 添加 green 类
$('#div1').removeClass('red');                      //从 div1 删除 green 类
```

与 addClass 和 removeClass 密切相关的还有一个 toggleClass 类，其作用为在添加和移除 CSS 类之间进行切换。如果元素已经应用了 CSS 类，则将其移除，否则添加。其用法如下：

```
$('#div2').toggleClass('test');
```

现在网页上经常见到一种表格的光棒效果，即鼠标移动到表格某行时，该行改变背景色高亮显示，鼠标移出时恢复成普通背景色，从而实现鼠标下的行始终高亮显示。可以使用 toggleClass()函数结合 mouseenter 和 mouseout 事件来实现这种效果。但是 jQuery 还提供了一个更方便的函数来实现这个功能，这个函数就是 hover()函数。hover()函数语法如下：

```
hover(
function() {/*这里写鼠标进入时要执行的代码*/},
function() {/*这里写鼠标移出时要执行的代码*/}
);
```

Hover()函数有两个参数，这两个参数都是函数，当鼠标进入元素时执行第 1 个函数，鼠标移出时执行第 2 个函数。利用 hover()函数结合 addClass 和 removeClass()函数，可以方便地实现表格光棒效果。

【例 10-2】 表格效果示例。

本例演示常用的表格效果。首先实现了光棒效果，随着鼠标移动，鼠标下的行总会高亮显示。另外，当单击某行时，还可以在选中和非选中之间进行切换。

（1）创建一个 ASP.NET Web 应用程序。

（2）在项目中添加 Northwind 数据库生成的实体数据模型。

（3）在项目中添加一个页面 GridEffect.aspx，在页面上放置一个 GridView，并在 Page_Load 中加载数据。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        //使用 LINQ 从实体框架中查询数据
        NorthwindEntities northwind = new NorthwindEntities();
        var query = from p in northwind.Products
                    select
                    new { p.ProductID, p.ProductName, p.QuantityPerUnit,
                    p.UnitPrice, p.UnitsInStock, p.ReorderLevel };
        //取查询结果前 10 条，并绑定到 GridView
        var list = query.Take(10).ToList();
        grid1.DataSource = list;
        grid1.DataBind();
    }
}
```

（4）在 GridEffect.aspx 页面中为 GridView 的选中和高亮行定义样式。

```
<style type="text/css">
.selected { background-color:#ddeeff; }                    /*选中样式*/
.highlight { background-color:#ffffee; }                   /*高亮样式*/
</style>
```

（5）在 GridEffect.aspx 页面中引入 jQuery。

```
<script src="jquery-1.4.2.min.js" type="text/javascript"></script>
```

（6）在 GridEffect.aspx 页面中编写 JavaScript 代码，实现选中样式和光棒效果。

```
<script type="text/javascript">
    $(function () {
        $('tr').click(                              //单击时切换选中样式
        function () {
            $(this).toggleClass('selected');
        });
        $('tr').hover(                              //鼠标划过时切换高亮样式
        function () {
            $(this).addClass('highlight');
        },
        function () {
            $(this).removeClass('highlight');
        }
        );
    }
    );
</script>
```

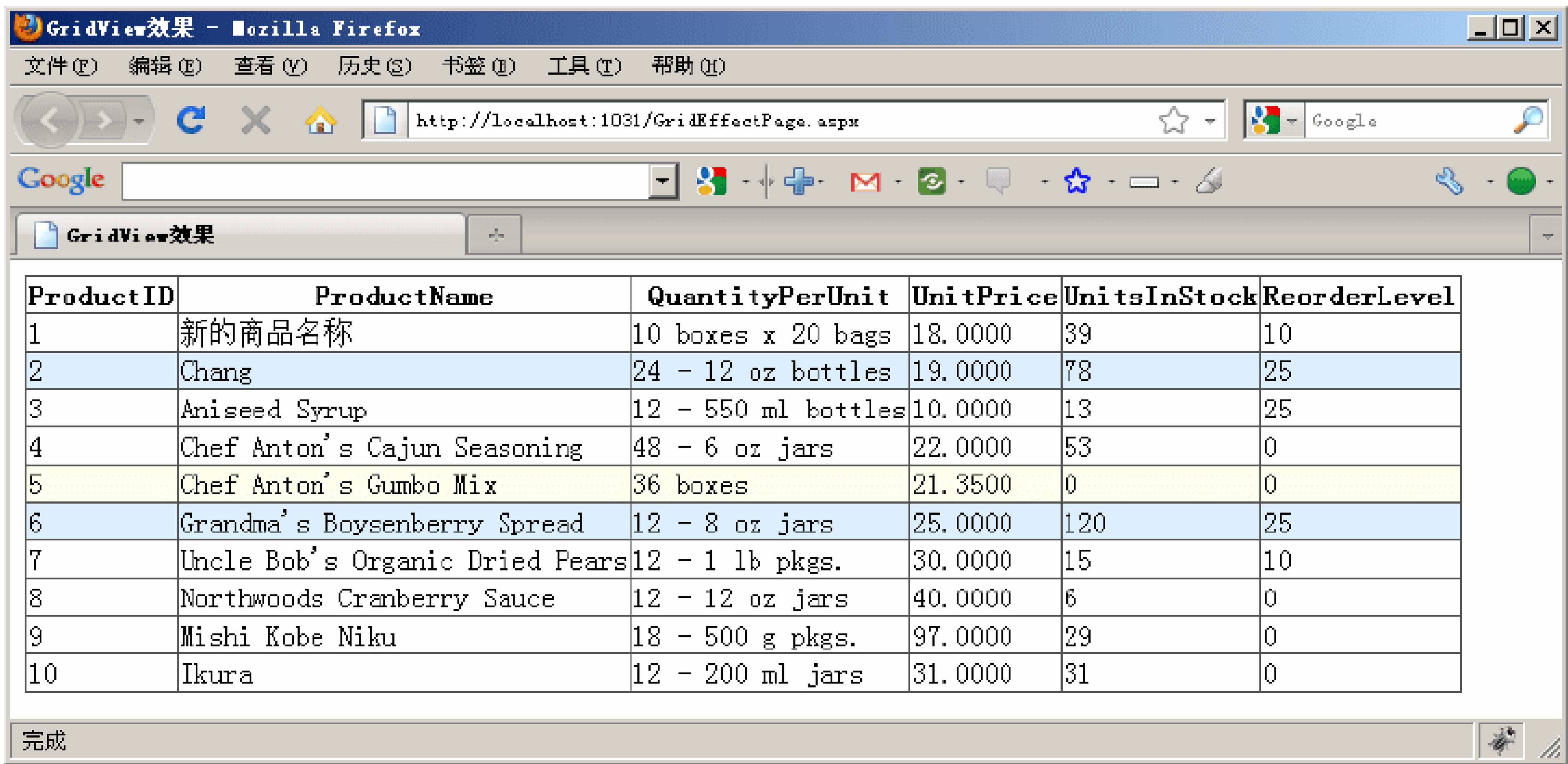
（7）运行 GridEffect.aspx 页面，运行界面如图 10.3 所示。

10.2.4　隐藏和显示元素

在制作网页时一个常用的视觉效果就是显示和隐藏一部分元素。使用 jQuery 的 hide() 和 show()方法可以实现这个功能。hide()和 show()方法的参数可以指定显示隐藏的速度，从

而达到合适的动画效果。hide()和 show()函数语法如下：

```
hide ( speed, callback );
show ( speed, callback );
```



ProductID	ProductName	QuantityPerUnit	UnitPrice	UnitsInStock	ReorderLevel
1	新的商品名称	10 boxes x 20 bags	18.0000	39	10
2	Chang	24 - 12 oz bottles	19.0000	78	25
3	Aniseed Syrup	12 - 550 ml bottles	10.0000	13	25
4	Chef Anton's Cajun Seasoning	48 - 6 oz jars	22.0000	53	0
5	Chef Anton's Gumbo Mix	36 boxes	21.3500	0	0
6	Grandma's Boysenberry Spread	12 - 8 oz jars	25.0000	120	25
7	Uncle Bob's Organic Dried Pears	12 - 1 lb pkgs.	30.0000	15	10
8	Northwoods Cranberry Sauce	12 - 12 oz jars	40.0000	6	0
9	Mishi Kobe Niku	18 - 500 g pkgs.	97.0000	29	0
10	Ikura	12 - 200 ml jars	31.0000	31	0

图 10.3　GridView 效果示例

两个函数实现相反的功能，具有相同的参数。第一个参数 speed 指定了显示或隐藏的动画速度，可以指定 slow、normal、fast 之一，也可以指定一个数值，表示完成显示或隐藏动画所需的时间。第二个参数 callback 是一个回调函数，当显示或隐藏完成时会调用此函数。这是一个可选参数。

【例 10-3】 显示和隐藏产品图片。

（1）添加一个 HTML 页面，在页面上放置一些静态控件以描述图书信息。

```
<div>
    书名：Design Patterns：Elements of Reusable Object-Oriented Software<br/>
    作者：Erich Gamma, Richard Helm, Ralph Johnson, John Vissides<br />
    出版社：机械工业出版社<br />
    说明：这是四人组(Gung of Four)写的最经典的设计模式著作，英文影印版<br />
    <a href="#" id="showImage">隐藏图片</a><br />
    <div id="bookImage">
    <img src="images/p1.jpg" alt="" style="width:200px" />
</div>
```

（2）在页面中导入 jQuery。

（3）在页面中编写代码，当单击“隐藏/显示图片”链接时，以普通的动画速度隐藏或者显示图书封面图片。

```
<script type="text/javascript">
    $(function () {
        $('#showImage').click(
        function () {
            //如果当前文字为“显示图片”，则显示图片，并将文字切换为“隐藏图片”
            if ($(this).text() == "显示图片") {
                $(this).text("隐藏图片");
                $('#bookImage').show('normal');
            }
            //如果当前文字为“隐藏图片”，则隐藏图片，并将文字切换为“显示图片”
```

```
            else {
                $(this).text("显示图片");
                $('#bookImage').hide('normal');
            }
        }
        );
    });
</script>
```

（4）运行此页面，运行效果如图 10.4 所示。

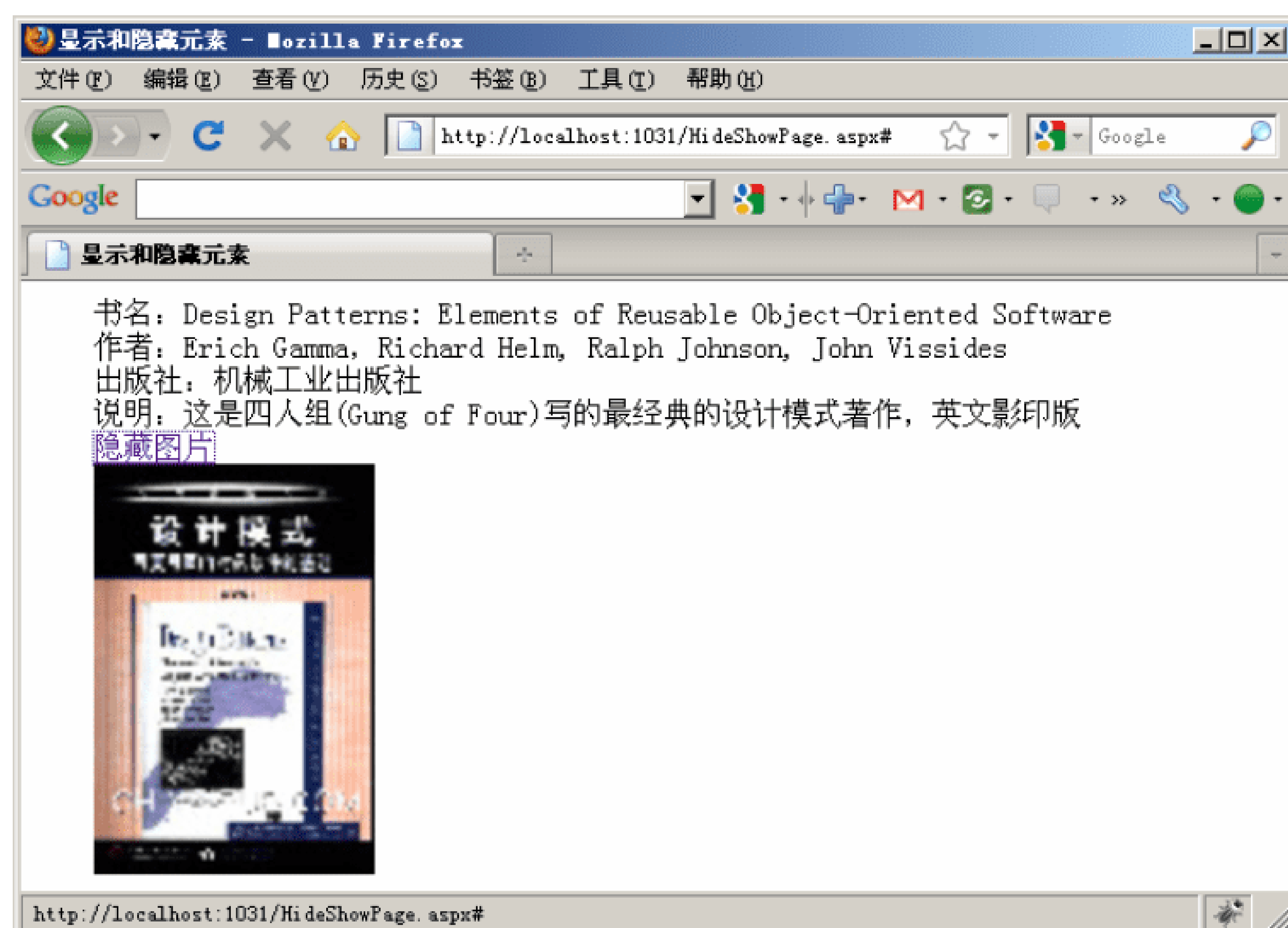


图 10.4　显示和隐藏图片

10.3　jQuery 常用选择器

jQuery 最基本的功能是操作 HTML 页面元素。jQuery 使用类似于 CSS 的选择器查找页面上的元素，然后进行各种操作，如更改样式、读取或设置文本、隐藏显示等。jQuery 支持多种选择器，常用有以下几种。

1．元素选择器

元素选择器能够选择某一类元素，如 a、img、input 等。例如，tr 表示选择所有 tr 元素，div 表示选择所有 div 元素。如果要给所有表格加上光棒效果，即随着鼠标移动始终突出显示鼠标下面的行，则可以使用以下代码实现。

```
$('tr').hover(
function () { $(this).addClass('highlight'); },
function () {$(this).removeClass('highlight');}
);
```

上述代码中$('tr')就选择了页面中所有 tr 元素，然后为其鼠标进入和移出事件分别编写

事件处理程序，添加和移除高亮显示样式。

2．ID选择器

ID 选择器是指根据一个元素的 ID 来选择元素。ID 选择器的语法是以#开头，紧接着是元素的 ID。例如$('#button1')将选择页面上 ID 为 button1 的元素。ID 选择器有时也和元素选择器一起使用，例如('div#main')选中 ID 为 main 的 div 元素。

3．类选择器

类选择器可以根据元素的 class 属性进行选择。类选择器的语法是以.开头，紧接着是 CSS 类名。例如，$('.important')将所有页面上所有应用了 important 类的元素。

4．后代选择器

如果在一个选择器后面有个空格，再跟另外一个选择器，则表示选择包含在第 1 个选择器中的第 2 个选择器。例如，“div p”选择 div 中出现的 p 元素。

5．子元素选择器

如果一个选择器后面是一个大于号>，后面再跟另外一个选择器，则表示选择直接包含在第 1 个选择器中的第 2 个选择器。注意，这种选择器与后代选择器的不同，后代选择器只要求后代（或者包含）关系，而不管直接还是间接，而子元素选择器只选择直接子元素。

10.4　jQuery+ASP.NET Web Service 实现 AJAX

在第 9 章中介绍了 AJAX 基本原理，使用 XMLHttpRequest 可以实现 AJAX 功能，但是语法较为烦琐。jQuery 对 XMLHttpRequest 进行了封装，可以更方便地与服务器进行交互，并取回响应结果。再结合 jQuery 强大的操作 DOM 元素的能力，将服务器返回结果及时显示在页面上，可以获得良好的用户体验和动态页面。

jQuery 的 ajax()函数封装了 AJAX 功能，ajax()函数语法如下：

```
$.ajax( 参数对象 );
```

其中参数对象是一个复杂的对象，包括许多属性，常用的有以下几个属性。

- ❑ type：HTTP 请求方法，可以是 GET 或者 POST。
- ❑ url：HTTP 语法的地址，在 ASP.NET 中，通常为一个 Web Service 或者 HttpHandler 地址。
- ❑ data：传递给服务器端的参数。
- ❑ success：ajax 调用成功后执行的回调函数。

【例 10-4】 验证用户名。

在用户注册时，需要验证新注册的用户名是否已经存在，如果已经存在则不允许再次注册。通过 AJAX 方式来完成这种验证，用户输入了用户后立即进行验证，而不需要用户

单击按钮，提高了用户体验。

（1）创建一个 ASP.NET Web 应用程序，在项目中添加一个 Web Service 用于检测用户名是否可以使用。Web Service 名称为 CheckUser.asmx，其中包含一个 canUse()方法，代码如下：

```
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
//若要允许从脚本中调用此 Web 服务，应添加下面这行代码
[System.Web.Script.Services.ScriptService]
public class CheckUser : System.Web.Services.WebService
{
    //用户名是否可以使用
    [WebMethod]
    public bool canUse(string user)
    {
        string[] existedUsers = { "zhang", "wang", "li", "zhao" };
                                                        //已经注册的用户
        for (int i = 0; i < existedUsers.Length; i++)
        {
            if (user == existedUsers[i])
                return false;
        }
        return true;
    }
}
```

提示： 如果需要用 JavaScript 调用 ASP.NET Web 服务，不管是用 jQuery，还是直接用 XMLHttpRequest，或者使用其他 JavaScript 框架，Web 服务类前都必须用 [System.Web.Script.Services.ScriptService]修饰，否则调用时就会出错。

（2）在项目中添加一个页面 UserCheckPage.aspx，页面上放置用于注册的各个控件。为简单起见，本例只放了两个 TextBox 控件，一个用于输入用户名，一个用于输入密码。

```
<form id="form1" runat="server">
<div>
<h3>用户注册</h3>
用户名：<asp:TextBox runat="server" ID="userid"></asp:TextBox>
<span id="checkUserResult"></span>
<br />
密码：<asp:TextBox runat="server" ID="password" TextMode="password">
</asp:TextBox><br />
省略用户注册其他信息...
</div>
</form>
```

（3）在 UserCheckPage.aspx 页面中引入 jQuery。

（4）在用户名控件的 blur 事件（失去焦点时发生）中，调用 Web Service 对输入的用户名进行检测，并显示相应提示。如果可以使用，则用绿色字体提示；如果不可以使用，

则用红色字体提示。

```
<script type="text/javascript">
    $(function () {
        $('#userid').blur(                          //用户名文本框失去焦点时
            function () {
                $.ajax({                            //AJAX 调用
                    type: 'POST',                   //使用 POST 方法
                    url: 'CheckUser.asmx/canUse',
                                        //被调用的 Web Service 地址和方法名
                    data: '{user:"' + $('#userid').val() + '"}',
                                        //传递给 Web Service 方法的参数
                    contentType: "application/json; charset=utf-8",
                    dataType: "json",       //使用 json 传递数据
                    success: function (result) {    //调用成功时执行此函数
                        if (result.d) { //若返回 true，则显示绿色提示
                            $('#checkUserResult').css('color', 'green');
                            $('#checkUserResult').html('此用户名允许使用');
                        }
                        else {          //若返回 false，则显示红色提示
                            $('#checkUserResult').css('color', 'red');
                            $('#checkUserResult').html('此用户名已经被注册，
                        不允许使用。');
                        }
                    },                  //success function 结束
                    error: function (){//错误发生时执行时函数
                        alert('Ajax 调用发生错误');
                    }                   //error function 结束
                }                       //$.ajax 参数结束
                );
            });                         //blur function 结束
    }                                   //最外层 function 结束
    );                                  //$结束
</script>
```

（5）运行 UserCheckPage.aspx 页面，输入用户名，再将输入焦点移动到密码框，则可以看到，会及时显示出用户名是否可用的提示，如图 10.5 所示。

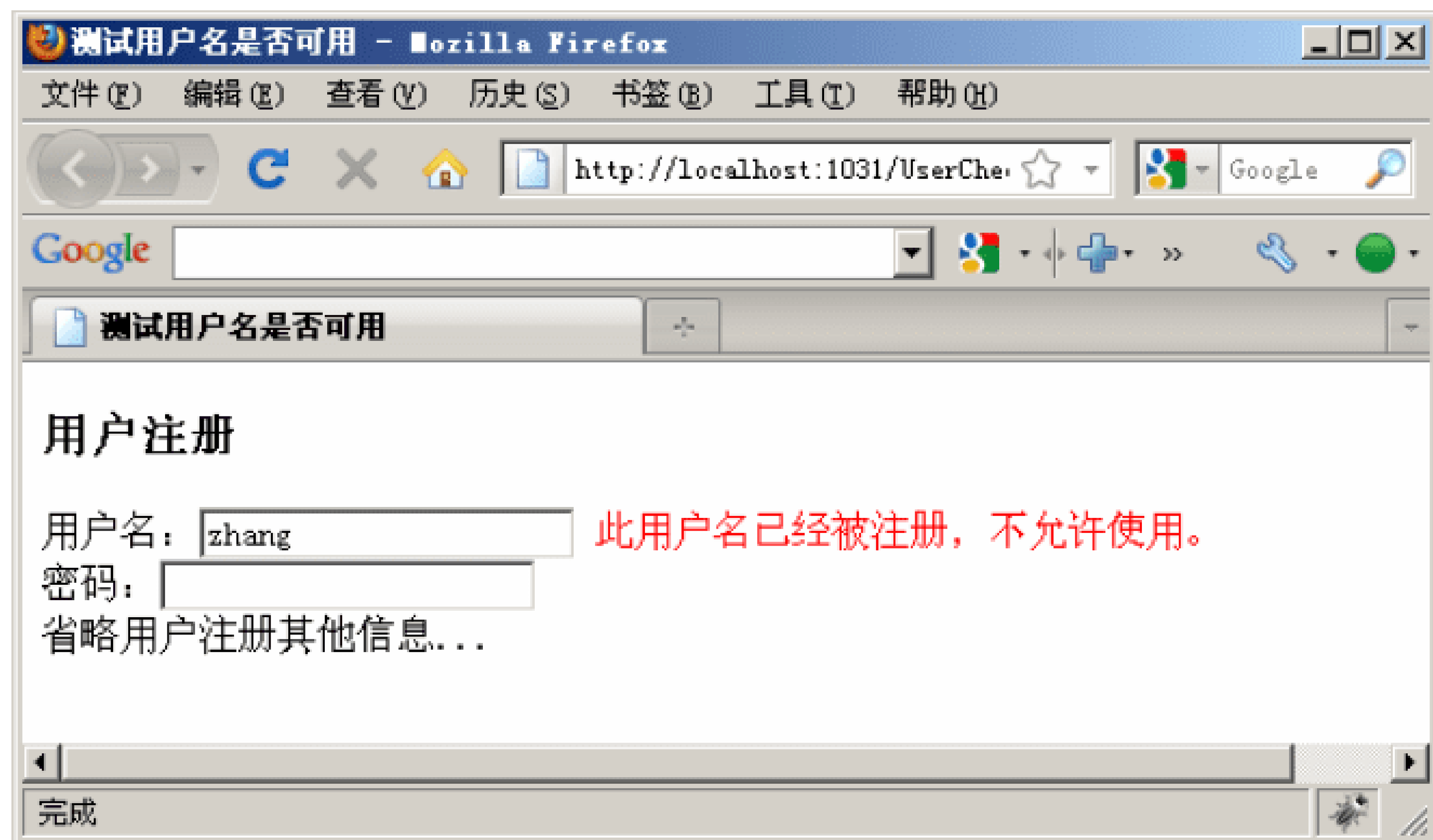


图 10.5　用户名检测页面

10.5　小　　结

本章介绍了一个当前很流行的 JavaScript 框架——jQuery。jQuery 的口号是“写得更少，做得更多”，能够大大降低 Web 开发人员的 JavaScript 编码工作量和难度。jQuery 使用类似于 CSS 的选择器查找 DOM 元素，方便灵活。jQuery 可以实现对 DOM 元素的样式修改、内容访问、添加、删除等操作，还可以 AJAX 方式与服务器交互，功能强大。本章仅对 jQuery 常见用法进行了介绍，如果需要深入理解 jQuery，请读者参阅专门的 jQuery 书籍或者官方网站。

第 3 篇　项目实战

第 11 章　通用权限管理系统

在各种类型的管理信息系统中，权限管理是一个普遍存在的问题。通常一个系统具有多类用户，每类用户具有不同的权限。例如在网上书店系统中，未登录用户可以浏览和搜索图书信息，登录的会员用户还可以查看自己的订单情况，系统管理员可以进入后台管理页面修改图书信息。本章将设计和实现一个简单的 ASP.NET 通用权限管理模块，此模块已经成功应用于作者开发的多个商用软件项目中。

11.1　整体设计思路

参照 Windows 中用户和权限管理的方式，结合 ASP.NET 应用程序的特点，本节所开发的通用权限管理基于角色进行权限分配，基于页面进行访问控制，基于 HttpModule 进行编码实现。本模块采用三层结构进行设计和开发。

11.1.1　需求分析

在一个应用程序系统中，根据权限的不同，用户可以分为若干类，例如有未注册用户、会员用户、系统管理员等。这种具有相同权限的用户类别称为一个角色。系统管理员可以添加各种角色，并为每种角色分配不同的权限。一个用户可以属于一种或几种角色，从而拥有所属角色的所有权限。

为了简化通用权限管理模块的编码实现，可以只考虑一个用户只属于一种角色的情况。如果一个用户确实属于多种角色，则可以为这个用户分别创建两个登录账号。例如张三既是普通员工，又是部门经理，拥有不同的权限，那么可以为张三创建两个登录账号 ZhangSan01 和 ZhangSan02，分别属于普通员工和部门经理的角色，张三可以根据具体情况选择一个用户账号进行登录。

在 ASP.NET 应用程序中，通常一个页面对应程序中的一个功能，例如，图书管理页面对应于图书信息编辑功能。如果一个用户或者角色拥有使用某个功能的权限，那么这个用户或者角色允许访问相应页面，否则不允许访问相应页面。根据这种思路，可以通过允许和禁止用户访问页面来实现权限控制。当用户要访问某一个页面时，首先判断此用户是否拥有访问此页面的权限，如果有则继续访问，如果没有则提示权限不足。

在实现应用中，还经常遇到这样一种情况，即有些页面允许所有用户访问，例如网上书店中的图书浏览和搜索页面。对于这种页面，不需要判断用户权限，可以直接访问。

11.1.2　数据库结构设计

根据前面的功能分析和设计思路，通用权限管理模块数据库中共包含 4 个表：用户表 User、角色表 User Role、功能模块表 ApplicationModule 和角色权限表 RoleRight，表中字段类型及含义如下。

1．角色表User

- ID：角色 ID，nvarchar(10)类型，主键。
- Name：角色名称，nvarchar(10)类型，不可空。
- Description：角色描述，nvarchar(50)类型，可空。

2．用户表User

- ID：用户 ID，登录时使用，nvarchar(10)类型，主键。
- Password：登录密码，nvarchar(20)类型，不可为空，默认值“123456”。
- UserNamer：用户名称，nvarchar(10)类型，不可为空。
- RoleID：用户所属角色 ID，外键关联到角色表 UserRole。

3．功能模块表ApplicationModule

- ID：模块 ID，nvarchar(10)类型，主键。
- Name：模块名称，nvarchar(30)类型，不可为空。
- URL：模块所对应的页面地址，nvarchar(50)类型，不可为空。
- Description：对于此功能模块的描述，nvarchar(80)类型。
- IsPublic：是否公共模块，bit 类型。公共模块表示所有用户都可以访问的模块。

4．角色权限表RoleRight

- RoleID：角色 ID，外键关联到 UserRole 表。
- ModuleID：模块 ID，外键关联到 ApplicationModule 表。
- TheRight：角色对模块的权限，nchar(1)类型，0 表示无权，1 表示有权。

以上各表之间关系如图 11.1 所示。

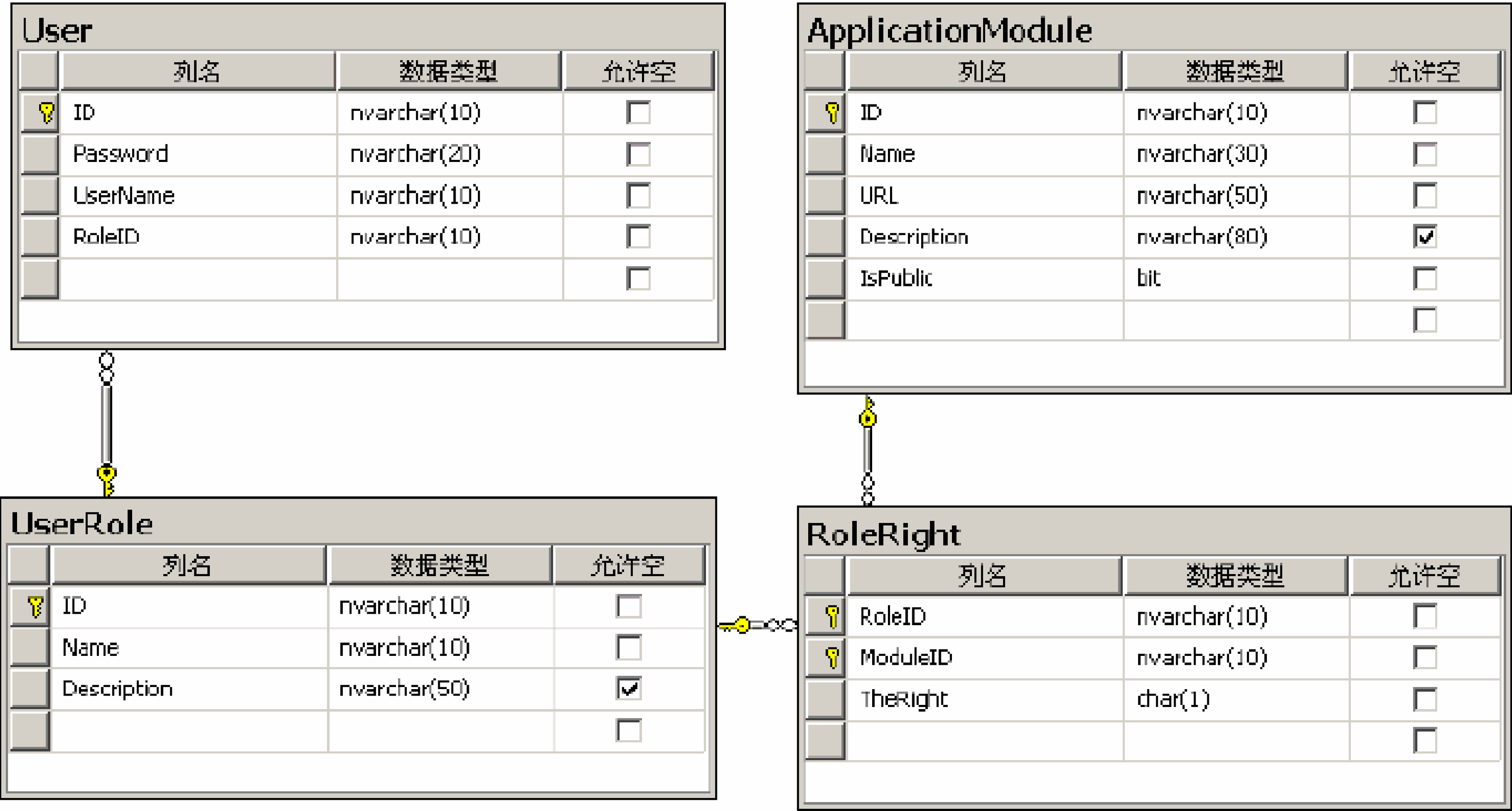


图 11.1　数据库结构

11.1.3　搭建项目框架

本节所设计开发的权限管理模块是一个通用模块，可以应用于不同需求的 ASP.NET 应用程序中。为了提高程序通用性，在 Visual Studio 中使用一个单独的解决方案实现通用权限管理模块。该模块采用三层结构设计，使用实体框架完成数据访问，解决方案中应该包含 4 个项目：实体框架层、数据访问层、业务逻辑层和 Web 表现层。搭建整个解决方案框架的步骤如下：

（1）创建一个空白解决方案。

（2）在解决方案中添加一个类库项目 SJL.Entity 作为实体框架层。

（3）在解决方案中添加一个类库项目 SJL.Dal 作为数据访问层。

（4）在解决方案中添加一个类库项目 SJL.Bll 作为业务逻辑层。

（5）在解决方案中添加一个 ASP.NET Web 应用程序项目 SJL.Web 作为表现层。

（6）在解决方案中添加一个类库项目 SJL.Common，此项目中包含会在三层用到的公共类。

（7）在各个项目之间添加引用关系，其他 3 个项目都引用实体框架层和公共类库，业务逻辑层引用数据访问层，表现层引用业务逻辑层。

11.2　公共类库和实体框架

根据 11.1 节的分析，通用权限管理系统采用三层结构，使用实体框架对业务实体类进行封装，并作为数据访问的基础。另外，项目中还有一个公共类库，其中包含一些公共代码。本节将介绍实体框架和公共类库的实现。

11.2.1　公共类库的实现

公共类库 SJL.Common 中包含了在数据访问层、业务逻辑层、表现层都会用到的类，这些类还会应用于除通用权限管理以外的其他项目中，所以这些类不能包含在通用权限管理项目的实体框架层中，而是要单独存储于一个类库，以便于在其他项目中复用。

公共类库中主要包含两个类：一个分页数据检索参数类 PageDataArgument，用于指定分页检索数据时的分页参数，如页号、页面大小等；一个日期范围类 DateRange，用于指定按照日期检索时的日期范围。PageDataArgument 类的代码如下：

```
//进行分页数据查询时的分页参数
public class PageDataArgument
{
   #region 字段
   public int pageIndex=0;                             //页码
   public int pageSize = 10;                           //页面大小
   public bool refreshCount = false;                   //是否更新记录总数
   public int count = -1;                              //记录总数
   private static PageDataArgument defaultArg = new PageDataArgument();
```

```
                                                      //默认分页参数
    private static PageDataArgument allDataArg
        = new PageDataArgument() {  pageSize = 99999 };
                                               //得到所有数据的分页参数
    #endregion
    #region 构造函数
    public PageDataArgument() { }
    public PageDataArgument(int index, int size, bool getCount)
    {
        pageIndex = index;
        pageSize = size;
        refreshCount = getCount;
    }
    #endregion
    #region 属性
    public static PageDataArgument allData
    { get { return allDataArg; } }
    public static PageDataArgument defaultValue
    {  get { return defaultArg; }}
    #endregion
}
```

日期范围类 DateRange 代码如下：

```
//日期范围类，用于指定查询时的起止日期
public class DateRange
{
    //以下两个字段定义了起始日期（含）和结束日期（含）
    public DateTime from;
    public DateTime to;
    ///<summary>
    ///根据两个日期（忽略时间）生成日期范围
    ///</summary>
    ///<param name="day1">起始日期</param>
    ///<param name="day2">终止日期</param>
    ///<returns></returns>
    ///<remarks>用户在查询时习惯于输入两个日期（不含时间），而查询时需要查询包括起
    ///止日期在内的时间范围。如输入 2010-1-1 和 2010-1-31，
    ///则得到的日期范围是 2010-1-1  00:00:00 至 2010-1-31  23:59:59
    ///这个方法能够方便地根据日期生成用户需要的时间范围
    ///</remarks>
    public static DateRange between2Date(DateTime day1, DateTime day2)
    {
        DateRange a = new DateRange();
        a.from = day1.Date;
        a.to = day2.Date.AddDays(1).AddSeconds(-1);
        return a;
    }
    //功能同上，根据两日期生成时间范围，但是两个日期参数是 string 类型
    public static DateRange between2Date(string day1, string day2)
    {
        return between2Date(DateTime.Parse(day1), DateTime.Parse(day2));
    }
}
```

11.2.2　实体框架层

本模块使用实体框架实现数据访问，构建实体框架层的步骤如下：

（1）打开实体框架层项目 SJL.Entity，从菜单中选择“项目”|“添加新项”命令，从弹出的“添加新项”对话框中选择“ADO.NET 实体数据模型”，命名为 UserRight。单击“确定”按钮，则弹出如图 11.2 所示的“实体数据模型向导”对话框，在其中选择“从数据库生成”模型内容。

（2）在“实体数据模型向导”对话框中单击“下一步”按钮，配置适当的数据库连接，再单击“下一步”按钮，出现如图 11.3 所示的“选择数据库对象”对话框。在此选中数据库中 4 个表，并选中“在模型中包含外键列”选项，然后单击“完成”按钮以完成实体数据模型向导。

（3）“实体数据模型”向导完成后，在项目中生成如图 11.4 所示的实体框架类。

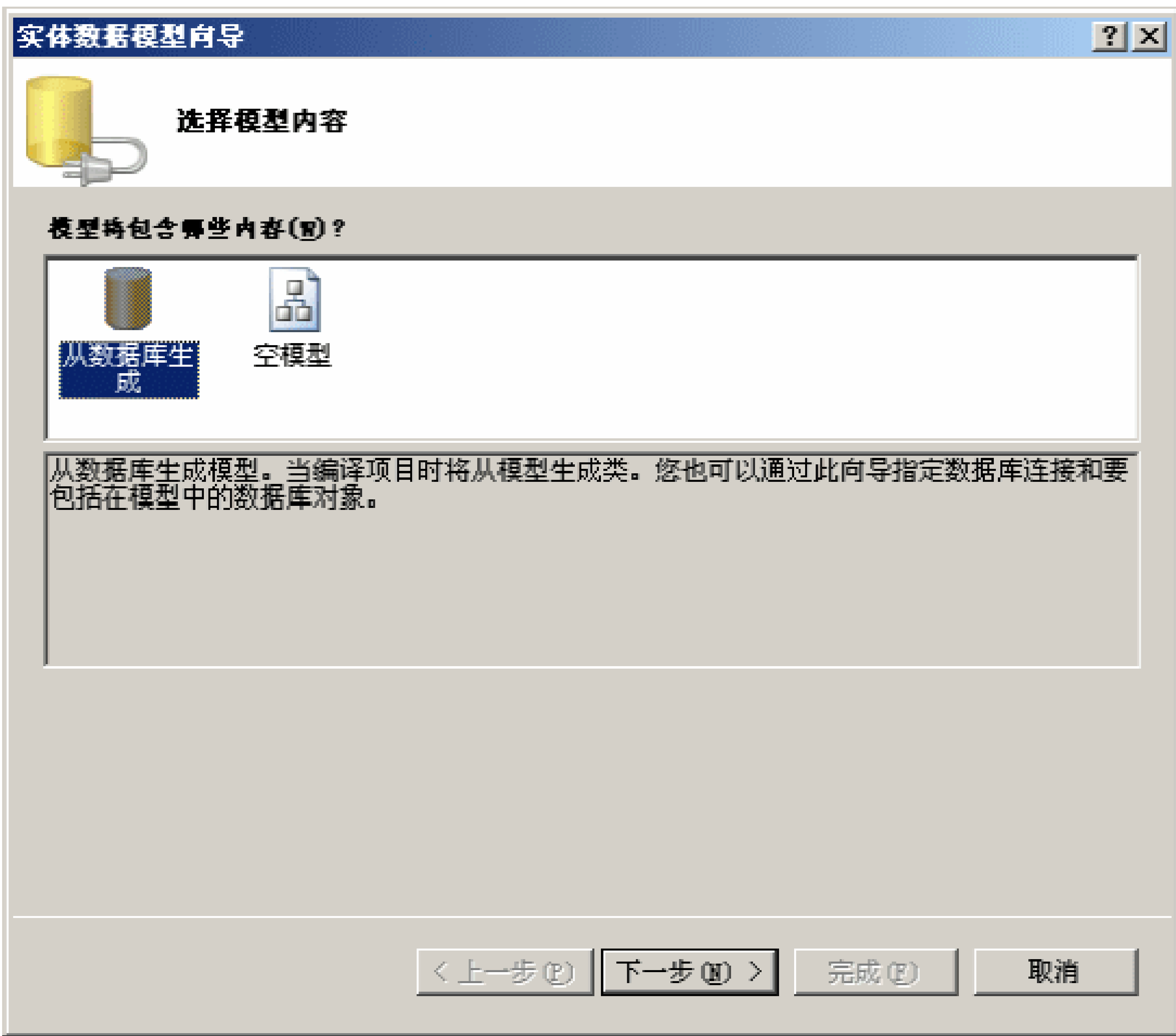


图 11.2　实体数据模型内容

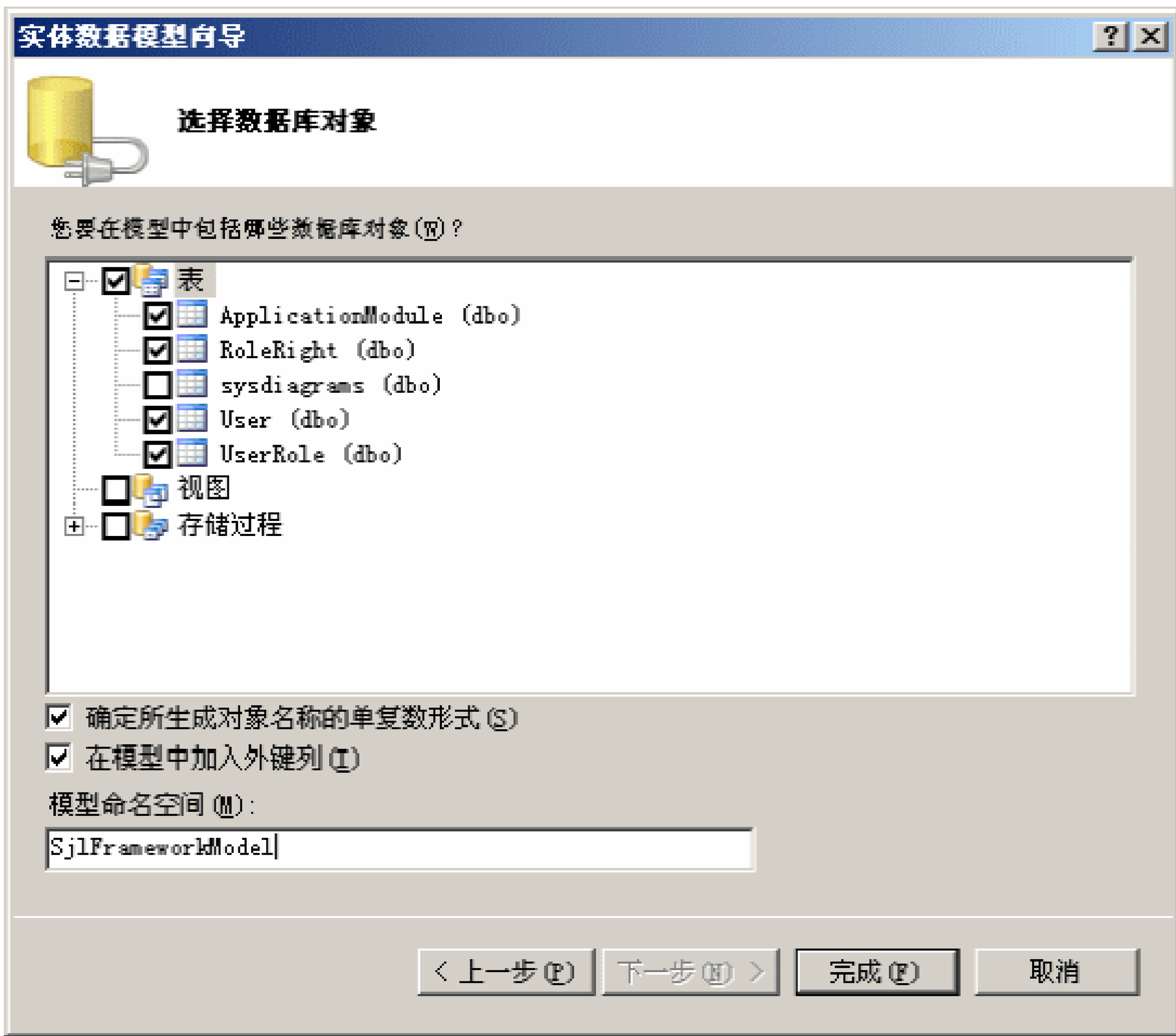


图 11.3　选择数据库对象

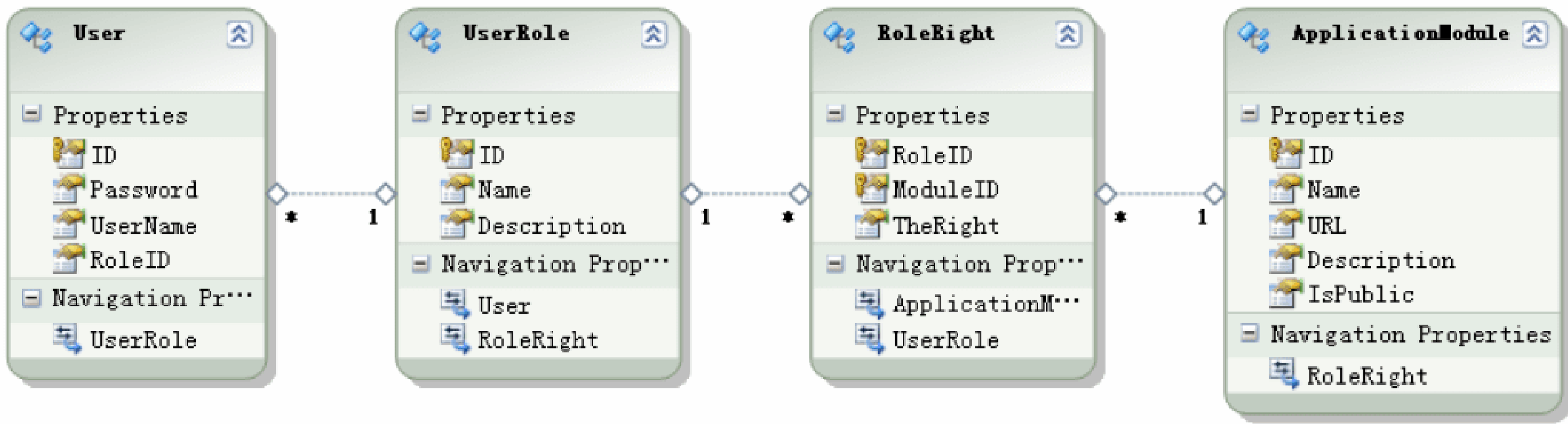


图 11.4　实体框架类

11.3　数 据 管 理

通用权限管理模块涉及到用户、角色、模块各方面的数据。用户可以对这些数据进行

浏览、修改、添加、删除等操作，本节将介绍这些功能的实现。

11.3.1　角色管理

角色管理功能可以实现添加、删除和修改用户角色。在三层结构的程序中，角色管理功能需要三层配合实现，具体步骤如下。

（1）在数据访问层 SJL.Dal 中添加一个类 UserRoleDal，在其中实现用户角色的增、删、改、查的功能，代码如下：

```
//用户角色数据访问层
public static class UserRoleDal
{
    /// <summary>
    /// 添加一个角色
    /// </summary>
    /// <param name="role">要添加的角色</param>
    /// <returns>添加的角色数</returns>
    public static int add(UserRole role)
    {
        return EntityUtility.add<UserRightContext, UserRole>(role);
    }
    /// <summary>
    ///根据 ID 删除一个角色
    /// </summary>
    /// <param name="id">要删除的角色 id</param>
    /// <returns>删除的角色数</returns>
    public static int delete(string id)
    {
        return EntityUtility.delete(getQuery(), m => m.ID == id);
    }
    //更新角色信息
    public static int update(UserRole role)
    {
        return EntityUtility.update<UserRightContext, UserRole>(role);
    }
    /// <summary>
    ///根据 ID 得到角色数据
    /// </summary>
    /// <param name="id">要查询的角色 id</param>
    /// <returns>查询到的角色数据</returns>
    public static UserRole getByID(string id)
    {
        return EntityUtility.selectOne(getQuery(), m => m.ID == id);
    }
    /// <summary>
    /// 分页得到角色列表
    /// </summary>
    /// <param name="page">分页参数</param>
    /// <returns>得到的指定页码的角色列表</returns>
    public static List<UserRole> getAll(PageDataArgument page)
    {
        return EntityUtility.selectMany(getQuery(), m => m.ID, page);
    }
    //得到用于执行角色数据访问的 ObjectQuery 对象
    private static System.Data.Objects.ObjectQuery<UserRole> getQuery()
```

```
    {
        return new SJL.Entity.UserRightContext().UserRole;
    }
}
```

（2）在业务逻辑层 SJL.Bll 中添加用于角色管理的类 UserRoleBll，由于此处没有业务逻辑，只需调用数据访问层的相应方法即可。

提示： 通用权限管理模块为本书第 1 个实用案例，故给出了完整代码。为节省篇幅，在以后的案例中，如果业务逻辑层只是调用数据访问层相应方法，本身没有实现其他功能，则书中不再给出业务逻辑层代码，读者可从本书配套光盘中得到完整代码。

```
public static class UesrRoleBLL
{
    public static int add(UserRole role)
    {
        return Dal0.add(role);
    }
    public static int delete(string id)
    {
        return Dal0.delete(id);
    }
    public static int update(UserRole role)
    {
        return Dal0.update(role);
    }
    public static UserRole getByID(string id)
    {
        return Dal0.getByID(id);
    }
    public static List<UserRole> getAll(SJL.Common.PageDataArgument page)
    {
        return Dal0.getAll(page);
    }
}
```

（3）在表现层中添加一个 ASP.NET 页面 UserRolePage.aspx 作为表现层，在页面顶部放一个 GridView 以显示角色列表，页面底部放几个 TextBox 以允许用户编辑和添加角色，页面代码如下：

```
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
    <h3>用户角色管理</h3>
     <%--在下面这个 Panel 中显示角色列表--%>
    <asp:Panel runat="server" ID="listPanel">
    <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns=
    "False"
         DataKeyNames="ID"
        CssClass="gridview" onrowcommand="GridView1_RowCommand" >
        <Columns>
            <asp:BoundField DataField="ID" HeaderText="角色编码" SortExpres-
            sion="ID">
            <HeaderStyle Width="100" />
            </asp:BoundField>
```

```
            <asp:BoundField DataField="Name" HeaderText="角色名称" SortExpres
            sion="Name">
            <HeaderStyle Width="120" /></asp:BoundField>
            <asp:BoundField DataField="description" HeaderText="角色描述"
            SortExpression="Name"/>
                <%--编辑和删除为模板列，显示一幅图片，并引发服务器端命令--%>
                <asp:TemplateField HeaderText="编辑">
                <ItemTemplate>
                <asp:ImageButton ID="ImageButton1" runat="server"  Command-
                Name="edit0" CommandArgument='<%#Eval("id") %>' ImageUrl=
                "../images/edit.png" />
                 </ItemTemplate>
                </asp:TemplateField>
                <asp:TemplateField HeaderText="删除">
                <ItemTemplate>
                <asp:ImageButton ID="ImageButton2"  runat="server" Command-
                Name="delete0"  OnClientClick="return confirmDelete();"
                CommandArgument='<%#Eval("id") %>' ImageUrl="../images/
                delete.png" />
                </ItemTemplate>
                </asp:TemplateField>
        </Columns>
    </asp:GridView>
        <webdiyer:AspNetPager ID="pager1" runat="server"
            onpagechanged="pager1_PageChanged">
        </webdiyer:AspNetPager>
    <br />
    <asp:Button ID="addButton" runat="server" Text="添加"
        onclick="addButton_Click"    /><br />
        </asp:Panel>
    <%--下面这个 Panel 用于角色编辑--%>
    <asp:Panel ID="editPanel" runat="server" >
    <table>
    <tr><td>角色编码</td><td>
        <asp:TextBox ID="moduleID" runat="server"></asp:TextBox>
        <asp:HiddenField ID="hiddenID" runat="server" />
        </td></tr>
    <tr><td>角色名称</td><td>
        <asp:TextBox ID="moduleName" runat="server"></asp:TextBox></td>
        </tr>
    <tr><td>角色描述</td><td>
        <asp:TextBox ID="description" runat="server"></asp:TextBox>
    </td></tr>
    <tr><td colspan="2">
        <asp:Button ID="saveButton" runat="server" Text="保存"
            onclick="saveButton_Click" />
        <asp:Button ID="cancelButton" runat="server"
            Text="取消" onclick="cancelButton_Click" />
    </td></tr>
    </table>
    </asp:Panel>
```

UserRolePage.aspx 页面设计外观如图 11.5 所示。

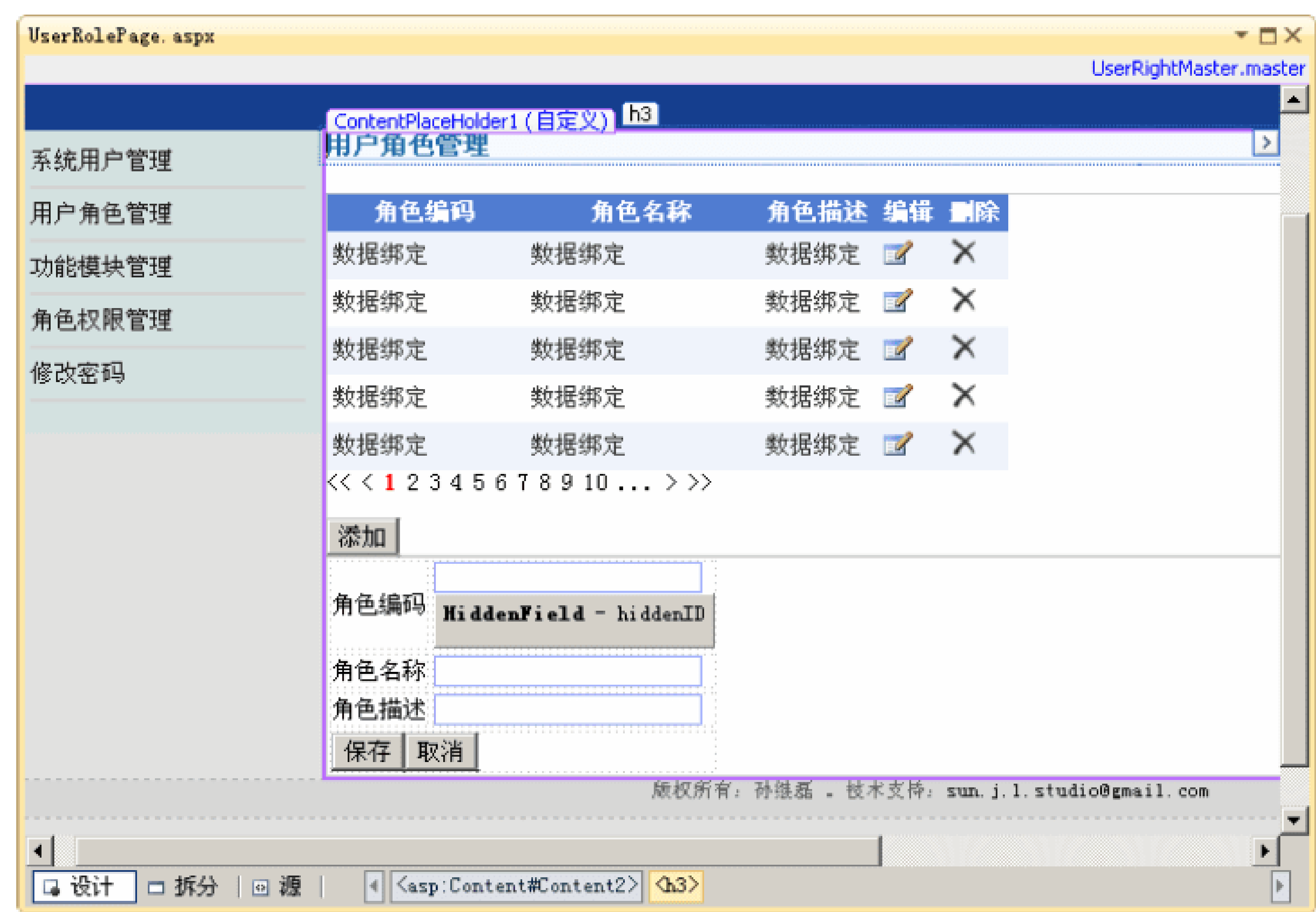


图 11.5　角色管理设计界面

（4）在 UserRolePage.aspx 页面中添加 JavaScript 代码，实现 GridView 鼠标光棒效果、删除提示、输入验证等功能。

```
<%--导入 CSS 文件和 JavaScript 文件--%>
<link  href="../css/ui-lightness/jquery-ui-1.7.2.css"  rel="stylesheet"
type="text/css" />
<script src="../js/jquery-1.3.2.js" type="text/javascript"></script>
<script src="../js/jquery-ui-1.7.2.js" type="text/javascript"></script>
<script src="../js/MyUtility.js" type="text/javascript"></script>
    <script type="text/javascript">
        //jQuery 初始化函数
        $(function () {
            //为 GridView 实现光棒效果
            $('table.gridview').find("tr").hover(
            function() { $(this).addClass('hoverRow'); },
                                               //鼠标进入时添加 hoverRow 样式
            function () { $(this).removeClass('hoverRow'); }
                                               //鼠标移出时移除 hoverRow 样式
            ); //$('table').tr.hover
            $SjlUtility.addButtonClass();    //为所有按钮应用外观样式
            $('#<%=saveButton.ClientID%>').click(checkInput);
                                               //为保存按钮添加输入验证
        });   //$(function)
        //删除确认函数
        function confirmDelete() {   return confirm("确实要删除吗？");     }
        //检测用户输入函数
        function checkInput() {
            if (checkInput2())
                return true;
            alert('必须输入完整的编码和名称。');
            return false;
        }
        function checkInput2() {
           //获取角色 ID 控件和角色名称控件的客户端 ID
            var userid = '#<%=roleID.ClientID %>';
            var userName = '#<%=roleName.ClientID %>';
            //获取角色 ID 控件和角色名称控件的文本
```

```
            var a=$(userid).val() ;
            if (a == "") return false;
            var b=$(userName).val();
            if ( b== '') return false;
            return true;
        }
    </script>
```

（5）在 UserRolePage.aspx 页面 Page_Load 事件中，显示第 1 页角色列表。

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindRoles();
        displyMode();
    }
}
//绑定一页角色数据到 GridView
private void bindRoles()
{
    //生成分页参数
    PageDataArgument arg = new PageDataArgument(pager1.CurrentPageIndex -
1, pager1.PageSize, true);
    var list = UserRoleBLL.getAll(arg);                    //执行查询
    pager1.RecordCount = arg.count;                        //刷新总数量
    GridView1.DataSource = list;                           //数据绑定
    GridView1.DataBind();
}
//进入只读模式，隐藏编辑区域
private void displyMode()
{
    editPanel.Visible = false;
    listPanel.Visible = true;
}
```

（6）当在角色列表中单击“编辑”按钮或者单击“添加”按钮时，则显示编辑区域。如果编辑已有角色，则将角色信息显示在编辑区域；如果添加新角色，则清空编辑区域内容。页面中定义了一个 string 类型的字段 newID 来标识是否有新记录。编辑、删除和添加按钮的代码如下：

```
private const string newID = "*NEWID*";
protected         void         GridView1 RowCommand(object         sender,
GridViewCommandEventArgs e)
{
    string id = e.CommandArgument.ToString();              //得到角色 ID
     //如果是删除命令，则删除当前角色，重新绑定数据
    if (e.CommandName == "delete0")
    {
        UserRoleBLL.delete(id);
        bindRoles();
    }
    //如果是编辑命令，则切换到编辑模式
    else if (e.CommandName == "edit0")
    {
        editMode(id);
    }
}
```

```
//当单击“添加”按钮时，切换到编辑模式
protected void addButton_Click(object sender, EventArgs e)
{
    editMode(newID);
}
/// <summary>
/// 进入编辑模式，显示编辑区域
/// </summary>
/// <param name="id">要编辑的角色 id，如果为 newID，则为添加新角色</param>
private void editMode(string id)
{
    editPanel.Visible = true;
    listPanel.Visible = false;
    hiddenID.Value = id;
    if (id == newID)
    {
        roleID.ReadOnly = false;          //如果为新增记录，则 ID 可修改
        clearEdit();
    }
    else
    {
        roleID.ReadOnly = true;           //如果编辑原有角色，则 ID 只读
        var module = UserRoleBLL.getById(id);
        roleID.Text = module.ID;
        roleName.Text = module.Name;
        description.Text = module.Description;
    }
}
```

（7）当用户单击“保存”按钮时，需要根据当前编辑的是新数据还是原有数据，相应地对数据库执行插入或者更新操作。保存按钮代码如下：

```
protected void saveButton Click(object sender, EventArgs e)
{
    UserRole role = new UserRole();
    role.Description = description.Text;
    role.ID = roleID.Text;
    role.Name = roleName.Text;
    if (hiddenID.Value == newID)            //如果 ID 为 newID，则添加记录
        UserRoleBLL.add(role);
    else                                    //否则更新记录
        UserRoleBLL.update(role);
    bindRoles();
    displyMode();
}
```

（8）在分页控件的 PageIndexChanged 事件中，重新绑定数据。

```
protected void pager1_PageChanged(object sender, EventArgs e)
{
    bindRoles();
}
```

（9）运行 UserRolePage.aspx 页面，运行界面如图 11.6 所示。

11.3.2　用户管理

用户管理页面可以实现用户的浏览、添加、删除、修改操作，但是不能修改其他用户

的登录密码。实现用户管理功能步骤如下。

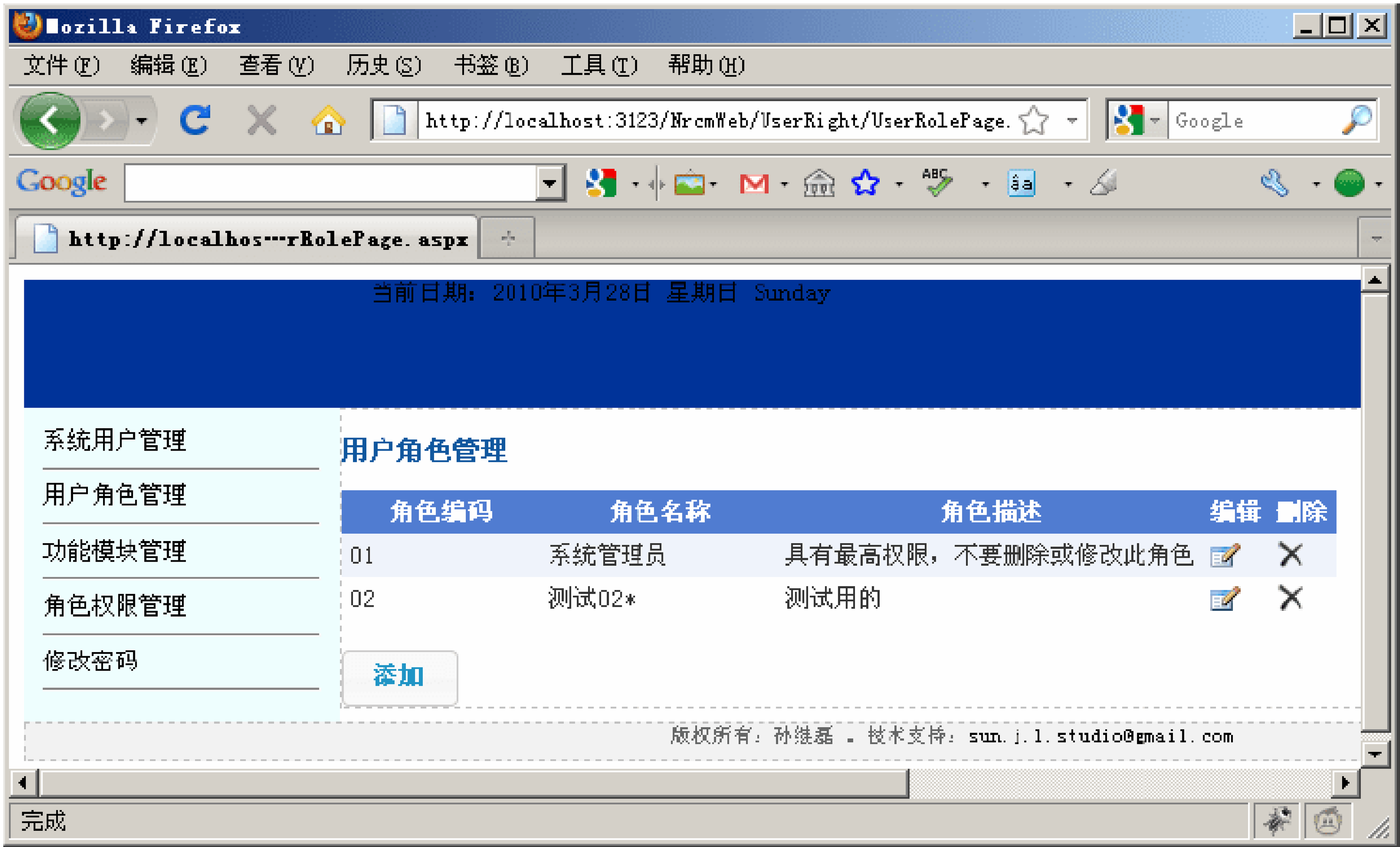


图 11.6　角色管理运行界面

（1）在数据访问层 SJL.Dal 中添加一个类 UserDal，并在其中实现对数据库中 User 表的增、删、改查功能。

```
public class UserDal
{
    //添加用户
    public static int add(User user)
    {
        return EntityUtility.add<UserRightContext, User>(user);
    }
    //删除用户
    public static int delete(string id)
    {
        return EntityUtility.delete(getQuery(), m => m.ID == id);
    }
    /// <summary>
    /// 更新用户信息，但不更新密码
    /// </summary>
    /// <param name="user">要更新的用户信息</param>
    /// <returns>被更新的用户数</returns>
    public static int update(User user)
    {
        return EntityUtility.update<UserRightContext, User>(user,
        "Password");
    }
    //根据用户 ID 得到用户信息
    public static User getByID(string id)
    {
        return EntityUtility.selectOne(getQuery(), m => m.ID == id);
    }
    /// <summary>
    /// 分页获取用户列表
    /// </summary>
    /// <param name="page">分页参数</param>
    /// <returns>指定页面用户列表</returns>
    public static List<User> getAll(PageDataArgument page)
    {
```

```
        var query = getQuery();
        var list= EntityUtility.selectMany(query, m => m.ID, page,false);
                                                          //加载用户列表
        list.ForEach(u => u.UserRoleReference.Load()); //加载用户角色
        query.Context.Dispose();
        return list;
    }
    public static bool isAdmin(User user)
    {
        return user.RoleID == "01";
    }
    public static int changePassword(User user)
    {
        return EntityUtility.update<UserRightContext, User>(user);
    }
    private static System.Data.Objects.ObjectQuery<User> getQuery()
    {
        return new UserRightContext().User;
    }
}
```

（2）在业务逻辑层 SJL.Bll 中添加一个 UserBll 类，实现与用户数据相关的业务规则。主要有两条业务规则：一是系统管理员用户 admin 不能被删除，二是添加新用户时设置默认密码 123456。UserBll 类的关键代码如下：

```
public static class UserBLL
{
    private const string ConstDefaultPassword = "123456";
    public static int add(User user)
    {
        user.Password = ConstDefaultPassword;        //添加用户时设置默认密码
        return UserDal.add(user);
    }
    public static int delete(string id)
    {
        //如果要删除 admin，则不允许，给出错误提示
        if (id.ToLower() == "admin")
            throw new ApplicationException("系统管理员用户 admin 不能被删除");
        return UserDal.delete(id);
    }
}
```

（3）在表现层 SJL.Web 项目中添加一个用户管理页面 ManageUser.aspx。页面布局与角色管理页面类似，页面代码如下：

```
<h3>系统用户管理</h3>
<asp:Panel runat="server" ID="listPanel">         <%--用户信息显示区域--%>
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ID"
    CssClass="gridview" onrowcommand="GridView1 RowCommand" >
    <Columns>
        <asp:BoundField DataField="ID" HeaderText="用户编码" >
        <HeaderStyle Width="100px" />
        </asp:BoundField>
        <asp:BoundField DataField="UserName" HeaderText="用户名称" >
        <HeaderStyle Width="120px" />
        </asp:BoundField>
        <asp:BoundField   DataField="Password"   HeaderText="   密   码   "
```

```
Visible="False" />
        <asp:TemplateField HeaderText="用户角色">
                                            <%--模板列，显示用户角色名称--%>
        <ItemTemplate>
        <%#Eval("UserRole.Name")%>
        </ItemTemplate>
        </asp:TemplateField>
        <%--模板列，以图形方式显示编辑按钮--%>
        <asp:TemplateField HeaderText="编辑">
        <ItemTemplate>
        <asp:ImageButton runat="server"  CommandName="edit0" Command-
        Argument='<%#Eval("id") %>' ImageUrl="../images/edit.png" />
        </ItemTemplate>
        </asp:TemplateField>
        <%--模板列，以图形方式显示删除按钮--%>
        <asp:TemplateField HeaderText="删除">
        <ItemTemplate>
        <asp:ImageButton  runat="server" CommandName="delete0"  OnClient-
        Click="return confirmDelete();"CommandArgument='<%#Eval("id") %>'
        ImageUrl="../images/delete.png" />
        </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
    <webdiyer:AspNetPager ID="pager1" runat="server"
        onpagechanged="pager1 PageChanged">
    </webdiyer:AspNetPager>
<br />
<asp:Button ID="addButton" runat="server" Text="添加"
    onclick="addButton_Click"   /><br />
    </asp:Panel>
<asp:Panel ID="editPanel" runat="server" >    <%--用户信息编辑区域--%>
<table>
<tr><td>用户编码</td><td>
    <asp:TextBox ID="userid" runat="server"></asp:TextBox>
    <asp:HiddenField ID="hiddenID" runat="server" />
    </td></tr>
<tr><td>用户名称</td><td>
    <asp:TextBox ID="userName" runat="server"></asp:TextBox></td></tr>
<tr><td>用户角色</td><td>
    <asp:DropDownList  ID="roleList"  runat="server"  DataTextField="name"
DataValueField="id">
    </asp:DropDownList>
</td></tr>
<tr><td colspan="2">
    <asp:Button ID="saveButton" runat="server" Text="保存"
        onclick="saveButton Click" />
    <asp:Button ID="cancelButton" runat="server"
        Text="取消" onclick="cancelButton_Click1" />
</td></tr>
</table>
</asp:Panel>
```

（4）在 ManageUser.aspx 页面的 Page_Load 事件中，绑定 GridView 用户列表和 DropDownList 角色列表。

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
```

```
    {
        initData();
        displyMode();
    }
}
//初始化数据，包括绑定用户列表、角色列表
private void initData()
{
    bindUsers();
    //获取角色列表并绑定到 DropDownList 控件中
    var roles = UserRoleBLL.getAll(SJL.Common.PageDataArgument.allData);
    roleList.DataSource = roles;
    roleList.DataBind();
    //在下拉表中添加一个“请选择”的项
    roleList.Items.Insert(0,new ListItem("--请选择--", "-1"));
}
//绑定用户列表
private void bindUsers()
{
    PageDataArgument arg = new PageDataArgument(pager1.CurrentPageIndex -
1, pager1.PageSize, true);
    var list = UserBLL.getAll(arg);
    pager1.RecordCount = arg.count;
    GridView1.DataSource = list;
    GridView1.DataBind();
}
```

（5）ManageUser.aspx 页面上编辑、删除、添加、保存按钮的代码与角色管理页面 UserRolePage.aspx 中的相应代码类似，此处不再给出具体代码。

（6）运行 ManageUser.aspx 页面，运行界面如图 11.7 所示。

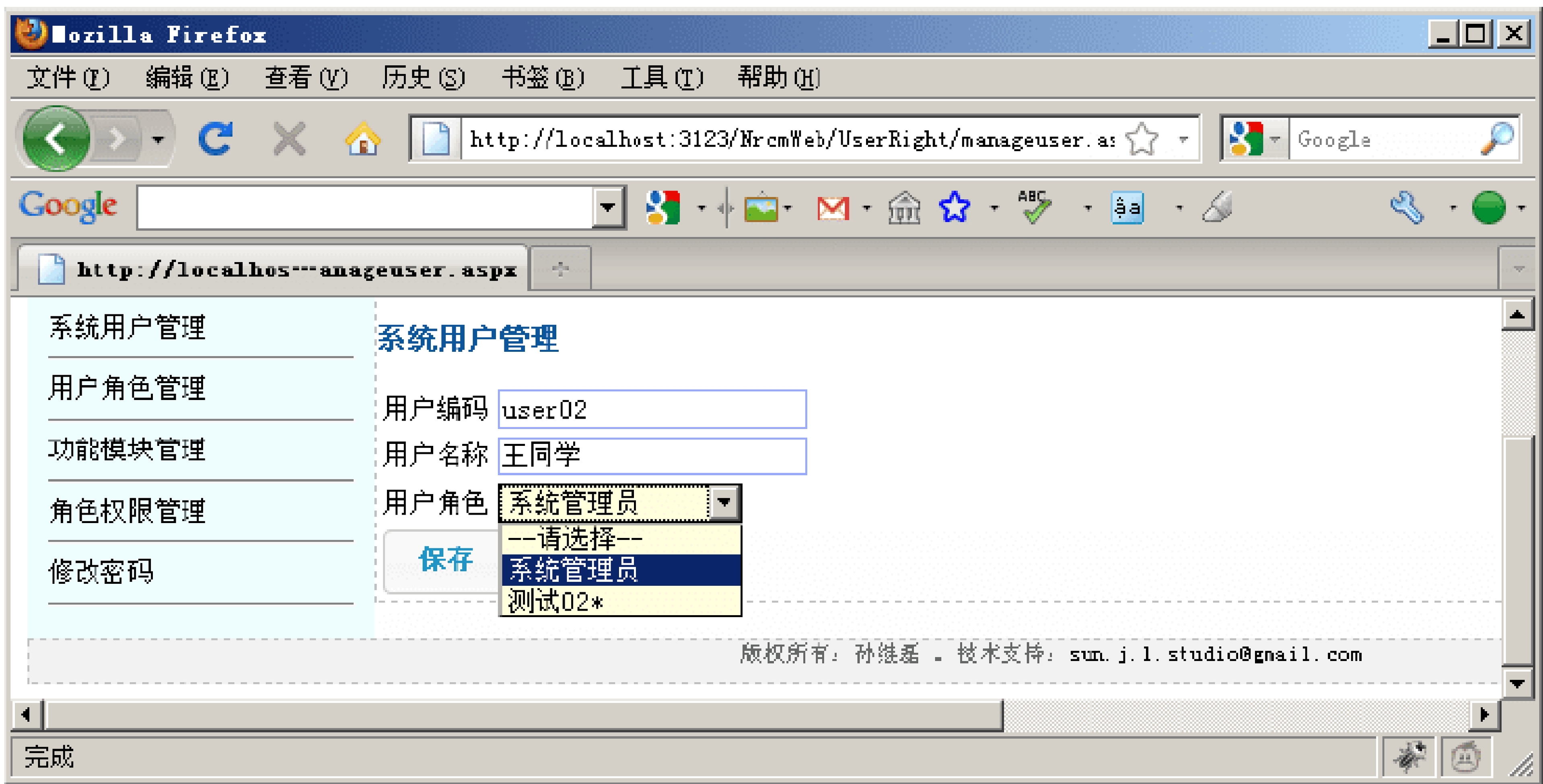


图 11.7　用户管理页面

11.3.3　功能模块管理

功能模块是权限分配的基本单位，在 ASP.NET 应用程序中，一个功能模块通常对应于一个 ASP.NET 页面。用户需要对功能模块进行浏览、添加、删除、修改等操作，这些功能的具体实现与前面所介绍的角色管理基本相同。当 Web 应用中有许多页面时，如果用户逐个添加这些页面，操作将非常繁琐，为此，可以编写一个根据网站目录下的 aspx 文件自动生成页面数据的方法，代码如下：

```
public static class AdminTool
{
    //根据网站中的页面自动生成模块信息
    public static void genereateModules()
    {
        string path = HttpContext.Current.Server.MapPath("~");
        generate( new DirectoryInfo(path));
    }
    static int n = 1;
    /// <summary>
    /// 递归方法，根据目录下的 aspx 文件生成模块信息
    /// </summary>
    /// <param name="directory">路径</param>
    private static void generate(DirectoryInfo directory)
    {
        FileInfo[] files = directory.GetFiles("*.aspx");
                                                //得到目录下的所有 aspx 文件
        foreach (var item in files)
        {
            //如果数据库不存在此页面，则添加
            if (ApplicationModuleBLL.getByUrl(item.Name) == null)
            {
                ApplicationModule m = new ApplicationModule();
                m.ID = string.Format("{0:00}", n++);
                m.Name = "自动生成的模块";
                m.Description = "自动生成的页面，没有描述。";
                m.URL = item.Name;
                ApplicationModuleBLL.add(m);
            }
        }
        DirectoryInfo[] dirs = directory.GetDirectories();
        foreach (var item in dirs)
        {
            generate(item);                     //递归进入下一级目录
        }
    }
}
```

11.3.4　角色权限管理

前面几个阶段建立了通用权限管理模块的基础数据，接下来需要为不同的角色分配权限，具体步骤如下：

（1）在数据库中添加一个存储过程以刷新角色权限矩阵。

```
CREATE PROCEDURE [dbo].[sp_refreshRoleRight]
AS
BEGIN
    set nocount on;
    --删除非法的角色数据
    delete from roleright where roleid not in(select id from userrole);

    --删除非法的模块数据
    delete from roleright where moduleid not in(select id from applica-
    tionmodule);
    --刷新角色权限矩阵
```

```
    insert into roleright(roleid,moduleid,theright)
    select r.id ,m.id,'0'
    from userrole r  cross join  applicationmodule m
    where not exists
     (select * from roleright rr where rr.roleid=r.id and rr.moduleid=m.id )
END
```

（2）在数据访问层 SJL.Dal 中添加一个类 RoleRightDal，实现对角色权限的增、删、改、查功能，关键代码如下：

```
public static class RoleRightDal
{
    //添加角色权限
    public static int add(RoleRight right)
    {
        return EntityUtility.add<UserRightContext, RoleRight>(right);
    }
    //更新角色权限
    public static int update(RoleRight right)
    {
        return EntityUtility.update<UserRightContext, RoleRight>(right);
    }
    //根据角色和模块获取权限信息
    public static RoleRight getByID(string role, string module)
    {
        var i= EntityUtility.selectOne(getQuery(), m => m.RoleID == role &&
        m.ModuleID == module);
        //loadDetail(i);
        return i;
    }
    //获取某角色的所有权限信息
    //参数：role 角色 ID，  page 分页参数
    public static List<RoleRight> getByRole(string role, PageDataArgument
    page)
    {
        var list= EntityUtility.selectMany(getQuery(), r =>r.ModuleID, page,
        m => m.RoleID == role);
        //list.ForEach(i => loadDetail(i));
        return list;
    }
    //刷新角色权限数据
    public static void refreshRoleRight()
    {
        var context = new UserRightContext();
        context.refreshRoleRight();
        context.Dispose();
    }
    /// <summary>
    ///用户是否有访问指定页面的权限
    /// </summary>
    /// <param name="role">角色 ID</param>
    /// <param name="page">请求访问的页面</param>
    /// <returns>是否有权限</returns>
    public static bool canAccessPage(string role, string page)
    {
        //根据 URL 获取对应的模块数据
        string module=ApplicationModuleDal.getByUrl(page).ID;
        //获取角色对该模板的权限，并判断是否可以访问
        var right = EntityUtility.selectOne(getQuery(), r => r.RoleID == role
```

```
&& r.ModuleID == module);
        return right.TheRight == "1";
    }
    private static System.Data.Objects.ObjectQuery<RoleRight> getQuery()
    {
        return new UserRightContext().RoleRight;
    }
}
```

（3）在业务逻辑层 SJL.Bll 中添加一个类 RoleRightBll。由于本例业务规则简单，此类的主要功能为向表现层提供接口，并调用数据访问层相应方法，为节省篇幅，此处省略 RoleRightBll 类的代码。

（4）在表现层 SJL.Web 中添加一个 RoleRightPage.aspx 页面，用于实现权限分配界面。页面用 GridView 列出了当前权限分配状态，并可以通过单击“允许”和“拒绝”按钮进行权限变更。

```
<h3>角色权限管理</h3>
选择角色：<asp:DropDownList ID="roleList" runat="server" DataTextField=
"name" DataValueField="id">
</asp:DropDownList>
<asp:Button ID="okButton" runat="server" Text="确定" onclick="okButton_
Click" /> <asp:HiddenField ID="hiddenRole" runat="server" /><br /><br />
<%--显示角色权限分配列表--%>
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    CssClass="gridview" onrowcommand="GridView1_RowCommand"  >
    <Columns>
        <asp:TemplateField HeaderText="模块编码">
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                       Text='<%# Eval("ApplicationModule.id") %>'></asp:
                       Label>
            </ItemTemplate>
            <ItemStyle Width="80px" />
            </asp:TemplateField>
           <asp:TemplateField HeaderText="模块名称">
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                  Text='<%# Eval("ApplicationModule.name") %>'></asp:Label>
            </ItemTemplate>
            <ItemStyle Width="100px" />
            </asp:TemplateField>
             <asp:TemplateField HeaderText="模块描述">
            <ItemTemplate>
                <asp:Label ID="Label3" runat="server"
                  Text='<%# Eval("ApplicationModule.description") %>'>
                  </asp:Label>
            </ItemTemplate>
            <ItemStyle Width="200px" />
            </asp:TemplateField>
             <asp:TemplateField HeaderText="有无权限">
            <ItemTemplate>
                <asp:CheckBox ID="CheckBox1" runat="server"
                  Checked='<%#Eval("theright").ToString().Trim()=="1" %>'
                  Enabled="false" />
            </ItemTemplate>
            </asp:TemplateField>
            <asp:TemplateField HeaderText="操作">
```

```
            <ItemTemplate>
            <asp:LinkButton runat="server" ID="grant" Text="允许" Command-
            Name="grant" CommandArgument='<%#Eval("ApplicationModule.
            id")%>'/>
            <asp:LinkButton runat="server" ID="deny" Text="禁止" Command-
             Name="deny" CommandArgument='<%#Eval("ApplicationModule.
             id")%>'/>
             </ItemTemplate>
            </asp:TemplateField>
    </Columns>
</asp:GridView>
<webdiyer:AspNetPager ID="pager1" runat="server"
    onpagechanged="pager1_PageChanged">
</webdiyer:AspNetPager>
```

RoleRightPage.aspx 页面设计外观如图 11.8 所示。



图 11.8　角色权限管理设计界面

（5）在页面的 Page_Load 事件中，将角色数据绑定到 DropDownList 控件。由于系统管理员角色拥有全部权限，所以不需要为其分配权限，从而在下拉列表中不需要显示系统管理员角色。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        RoleRightBLL.refreshRoleRight();
        initRoles();
    }
}
private void initRoles()
{
    var list = UserRoleBLL.getAll(PageDataArgument.allData);
                                                //得到所有角色列表
    //系统管理员具有全部权限，不需要分配，所以从列表中删除系统管理员角色
    list.Remove(list.Find(r => r.ID == "01"));
    roleList.DataSource = list;
    roleList.DataBind();
    if (roleList.Items.Count == 0)
    {
        ClientScript.RegisterStartupScript(this.GetType(),
```

```
"addrolefirst",
            "<script>数据库中没有角色。请先添加角色再为其分配权限。</script>");
        return;
    }
    roleList.SelectedIndex = 0;
    bindRights();
}
```

（6）在“确定”按钮的 Click 事件中，根据用户所选择的角色列出角色权限分配情况。

```
//显示角色权限列表
private void bindRights()
{
    if (roleList.SelectedIndex < 0) return;          //如果未选择角色则返回
    string role = roleList.SelectedValue;            //得到角色 ID
    hiddenRole.Value = role;
    //创建分页查询参数
    PageDataArgument page = new PageDataArgument();
    page.refreshCount = true;
    page.pageSize = pager1.PageSize;
    page.pageIndex = pager1.CurrentPageIndex - 1;
    //执行查询，将查询结果绑定到 GridView
    var list = RoleRightBLL.getByRole(role,page);
    pager1.RecordCount = page.count;
    GridView1.DataSource = list;
    GridView1.DataBind();
}
protected void okButton Click(object sender, EventArgs e)
{
    pager1.CurrentPageIndex = 1;
    bindRights();
}
```

（7）当用户单击列表中的“允许”或“禁止”按钮时，会触发 GridView 的 RowCommand 事件，需要在此事件中执行相应的权限改动，代码如下：

```
protected void GridView1_RowCommand(object sender, GridViewCommandEvent-
Args e)
{
    string newRigth="0";
    //根据用户选择的是允许还是禁止，设置新权限的值
    if (e.CommandName == "grant")
        newRigth = "1";
    else if (e.CommandName == "deny")
        newRigth = "0";
    else
        return;
    string id = e.CommandArgument.ToString();
    //更新权限数据
    RoleRight right = new RoleRight();
    right.ModuleID = id;
    right.TheRight = newRigth;
    right.RoleID = hiddenRole.Value;
    RoleRightBLL.update(right);
    bindRights();                                          //重新绑定权限列表
}
```

（8）运行 RoleRightPage.aspx 页面，运行界面如图 11.9 所示。

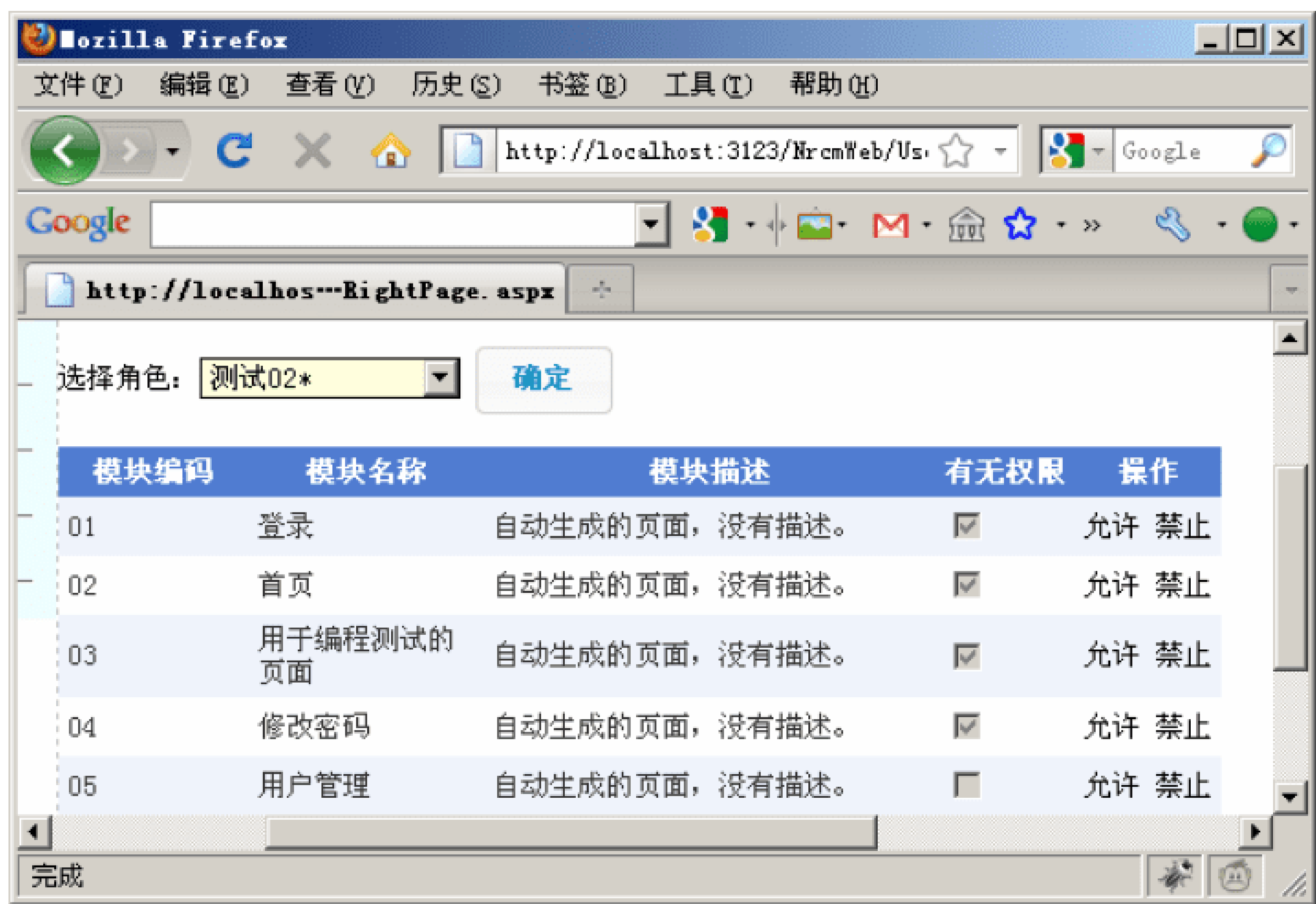


图 11.9　权限分配页面

11.4　权 限 控 制

本节将介绍通用权限管理系统核心功能的实现，即权限检测与控制。在具有权限管理功能的 ASP.NET Web 应用中，当用户要访问某一个页面时，首先判断此用户是否拥有访问所请求页面的权限，如果有则继续访问，如果没有则提示权限不足。可以在页面的 Page_Load 事件中对用户权限进行检测，但是由于一个 ASP.NET 应用程序中会有多个页面，这种方案需要在每个页面中都编写代码，非常不方便。这个问题还存在一种更好的解决方案，即 HTTP 模块（HttpModule）。

11.4.1　用户权限检测

在 ASP.NET 应用程序中，每个 HTTP 请求发生时，都会执行已经注册的 HttpModule 相应方法。因此，在 HttpModule 中适合编写所有页面的公共代码。在通用权限管理系统中，每个页面被请求时，都需要检测用户权限是否合适，可以把用户权限检测代码写到 HttpModule 中。

进行用户权限检测的过程如下：用户请求一个 URL（如 http://localhost/book.aspx?id=2），程序先取出 URL 中包含的文件名（如 book.aspx），文件名通常为 URL 中最后一个“/”号后面和“?”号前面的部分，然后再判断此文件是否需要受权限保护。如果受权限保护，则要根据当前登录的用户角色判断用户是否拥有访问此文件的权限。如果权限检测通过，则允许继续访问指定资源，否则跳转到登录页面。图 11.10 描述了权限检测流程。

实现用户权限检测和控制的具体步骤和代码如下：

（1）在表现层项目 SJL.Web 中添加一个 ASP.NET 模块，命名为 CheckUserModule。

（2）在 CheckUserModule 的 Init 事件中为 HTTP 应用的 AcquireRequestState 事件注册一个事件处理程序。

```
public void Init(HttpApplication context)
{
    context.AcquireRequestState += new EventHandler(checkUserRight);
}
```

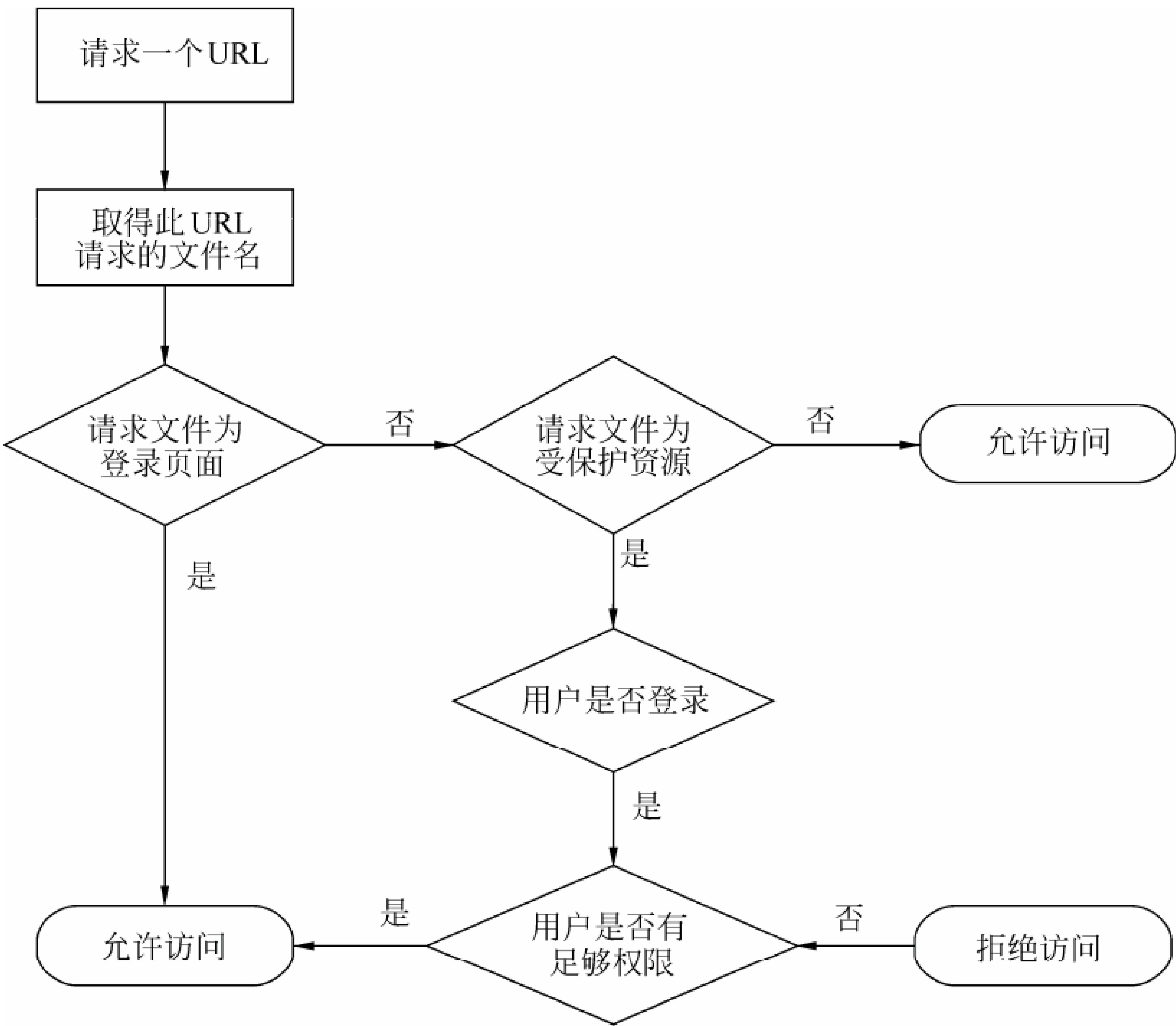


图 11.10　权限检测流程

（3）在 checkUserRight()方法中，实现图 11.10 所描述的检测流程。

```
/// <summary>
/// 检测用户权限
/// </summary>
void checkUserRight(object sender, EventArgs e)
{
    HttpApplication application = (HttpApplication)sender; //获取应用程序
    string url = HttpContext.Current.Request.Url.ToString();  //获取 URL
    int start=url.LastIndexOf('/') + 1;     //查找 URL 中最后一个/的位置
    int end=url.IndexOf('?',start);         //查找 URL 中？位置
    string requestPage = null;
    if (end < 0) end = url.Length - 1;
    requestPage=url.Substring(start, end - start +1);  //得到所请求的页面
    requestPage = requestPage.ToLower();
    if (requestPage == loginPage) return;
    if (!isProtectedResource(requestPage)) return;
    User user=SJL.Web.HttpCode.WebUtility.currentUser; //获得当前用户
    //如果当前用户为空，则转到登录页面
    if (user==null)
    {
        application.Response.Redirect("~/Login.aspx");
        return;
    }
    //如果当前用户为系统管理员，则不需要验证权限，直接返回
```

```
    if (SJL.Bll.UserRight.UserBLL.isAdmin(user)) return;
    //检测用户权限
    if        (!SJL.Bll.UserRight.RoleRightBLL.canAccessPage(user.RoleID,
requestPage))
        application.Response.Redirect("~/AccessDeny.htm");
}
/// <summary>
/// 判断页面是否为受权限管理保护的资源（如 Aspx 等）
/// </summary>
/// <param name="page">所请求的页面</param>
/// <returns>是否受保护</returns>
bool isProtectedResource(string page)
{
    page = page.ToLower();
    string[] protectedFiles = {".aspx",".asmx",".ashx"};//受保护资源的扩展名
    for (int i = 0; i < protectedFiles.Length; i++)
    {
        if(page.EndsWith(protectedFiles[i]))
            return true;
    }
    ApplicationModule module = SJL.Bll.UserRight.ApplicationModuleBLL.
    getByUrl(page);
    if (module == null) return true;
    return !module.IsPublic;                    //如果页面为公共模块则不受保护
}
```

上述代码中用到了 WebUtility 类的一个属性 currentUser，该属性用于读取和设置当前 Session 中保存的用户信息，代码如下：

```
public static class WebUtility
{

    private const string SessionUser = "thisuser";             //登录用户
    public static SJL.Entity.User currentUser
    {
        get
        {
            if (HttpContext.Current == null) return null;
            return HttpContext.Current.Session[SessionUser] as global::
            SJL.Entity.User;
        }
        set
        {
            HttpContext.Current.Session[SessionUser] = value;
        }
    }
}
```

11.4.2　用户登录

在用户权限检测模块中，如果用户不具有访问所请求资源的权限，则用户通常需要跳转到登录页面以合适的用户身份登录，再访问此前所请求的资源。登录页面除了验证用户名和密码的正确性以外，还将用户登录信息保存到 Session 中，以方便在整个应用程序中使用。登录页面 Login.aspx 布局和功能都很简单，代码如下：

```
<div id="login">
       用户名 <asp:TextBox ID="userName" runat="server" ></asp:TextBox>
    密 码
    <asp:TextBox ID="password" runat="server" TextMode="Password" >
    </asp:TextBox>
    <asp:Button ID="login2" runat="server" Text="登录" Width="80" Height=
    "25" onclick="login2 Click"/>
</div>
protected void login2 Click(object sender, EventArgs e)
{
   //获取用户输入的用户名和密码
   string u = userName.Text;
   string p = password.Text;
   var user = UserBLL.getByID(u);             //根据用户名得到用户信息
   //若密码正确，则将用户保存到 Session，跳转到首页
   if (user.Password == p)
   {
      WebUtility.currentUser = user;
      Response.Redirect("~/Default/Default.aspx");
   }
   //若密码不正确，则用 JavaScript 显示提示消息
   else
   {
      ClientScript.RegisterStartupScript(this.GetType(), "loginerror",
      "<script>alert('用户名或密码有误，登录失败！');</script>");
   }
}
```

11.5　小　　结

本章设计开发了一个 ASP.NET 通用权限管理系统，可以实现基于页面的简单权限控制。此权限管理系统可以不加修改地用于各个 ASP.NET 应用程序，只需要在目标应用程序中注册 HttpModule，并添加相应的数据（包括用户、角色、功能模块、权限）即可。

第 12 章　县长公开电话受理系统

本章所介绍的案例是作者为某县政府开发的一个软件。出于安全和隐私方面的原因，书中不便给出此县的真实名称，下文均以 A 县代替。为了更加及时地了解群众所关心的问题，及时为群众排忧解难，A 县政府设立了书记、县长公开电话。为了准确高效地管理公开电话信息，跟踪监控问题处理情况，特开发了本软件系统。

12.1　整体设计思路

本节将介绍 A 县书记、县长公开电话业务处理流程，分析公开电话受理系统项目的需求，提出此系统整体设计思路，实现数据库结构设计。

12.1.1　需求分析

A 县政府设立了书记、县长公开电话，群众可以随时拨打此电话反映问题。公开电话受理中心有数名工作人员，负责接听群众电话，记录群众所反映的问题，并将问题提交给相关责任人。公开电话受理中心的工作人员需要跟踪群众所反映问题的整个处理过程，并对数据进行一定的统计分析。公开电话业务处理大致流程如下：

（1）群众拨打公开电话，受理中心工作人员接听并记录群众所反映的问题。在本系统中，群众通过公开电话所反映的问题称为“事件”。

（2）公开电话受理中心人员对群众所反映的事件进行初步分析，确定事件的重要级别，并做出不同处理。如果所反映问题较小，则受理中心工作人员可予以解答，处理过程到此结束。如果问题较大，则需要根据问题反映的具体内容及所涉及的相关部门，将问题转交给县领导或者责任单位（如教育局、卫生局、环保局等）进行处理。

（3）群众所反映的事件由公开电话受理中心转交给县长或者其他责任单位后，相关责任人需要及时解决问题，并随时将进展情况通报给公开电话受理中心。

（4）一个事件处理结束后，公开电话受理中心工作人员需要对当初反映此事件的群众进行回访，调查群众对于事件处理结果的满意程度。

（5）公开电话受理中心人员需要对公开电话受理事件数据进行汇总，形成报表。

根据以上业务流程的分析，可以确定公开电话受理系统主要有以下需求：

- ❑ 用户管理和权限管理。
- ❑ 记录群众通过公开电话所反映的事件信息。
- ❑ 跟踪公开电话所受理事件的处理进展，并及时更新事件状态。

- ❑ 打印公开电话处理过程中产生的各种单据。
- ❑ 按照各种条件查询公开电话受理事件情况。
- ❑ 产生各种统计报表。

12.1.2　数据库结构设计

本系统最为核心的数据为公开电话受理事件详情，如受理时间、反映人、反映内容等。与此相对应，数据库中用于存储此数据的表 EventInfo 最为重要，结构也较复杂，其他表结构都较为简单。EventInfo 表中各字段的含义如下：

- ❑ EventID：事件编号，nvarchar(10)类型，主键。
- ❑ HandleDate：事件受理日期，datetime 类型，默认值为当前日期。
- ❑ HandleUser：受理人员，nvarchar(10)类型。
- ❑ EventSource：事件来源，表示来源于书记公开电话、县长公开电话、县长信箱等，int 类型，外键关联到 EventSource 表。
- ❑ PersonName：反映人姓名，nvarchar(10)类型。
- ❑ PersonPhone：反映人联系电话，nvarchar(20)类型。
- ❑ PersonAddress：反映人地址，nvarchar(50)类型。
- ❑ TypeID：事件类型，表示所反映问题属于哪一类（如环保、教育、医疗等），int 类型，外键关联到 EventType 表。
- ❑ EventTitle：事件主题，nvarchar(50)类型。
- ❑ EventContent：事件内容，nvarchar(1000)类型。
- ❑ StatusID：事件处理状态，如办结、待办等，int 类型，外键关联到 EventStatus 表。
- ❑ EventResult：事件处理结果，nvarchar(1000)类型。
- ❑ EvaluationID：群众满意程度，int 类型，外键关联到 EventEvaluation 表。
- ❑ ResponsibleDepartment、ResponsibleDepartment2、ResponsibleDepartment3：第 1、第 2 和第 3 责任单位，nchar(2)类型，外键关联到 Department 表。
- ❑ ResponsiblePerson：责任人，nvarchar(10)类型。
- ❑ FinishDate：事件完成时间，datetime 类型。
- ❑ ApproveLeader：签批领导，nvarchar(10)类型。

数据库中其他各表结构如图 12.1 所示。

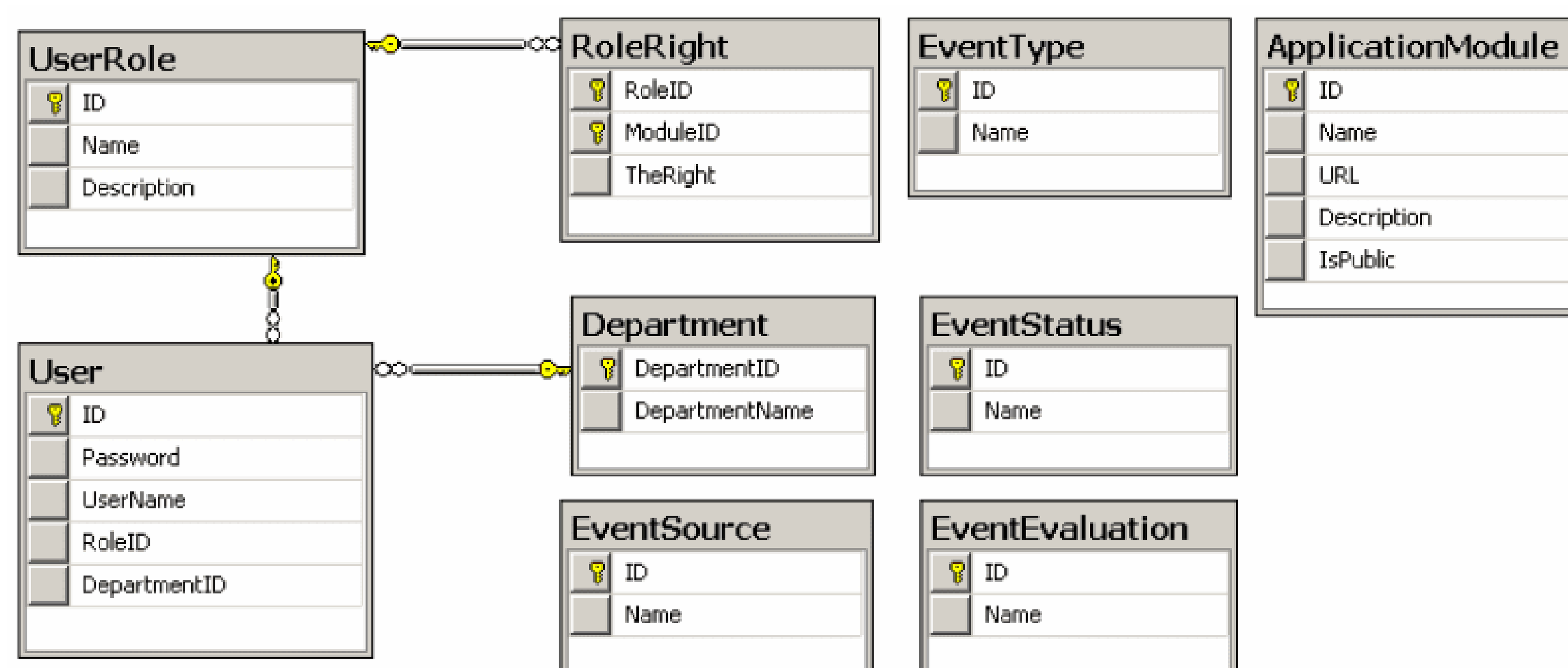


图 12.1　数据库结构

12.1.3　搭建项目框架

本项目采用多层结构设计，使用实体框架完成数据访问。解决方案中应该包含以下几个项目：实体框架层、数据访问层、业务逻辑层、Web 表现层、公共类库、单元测试项目、数据库管理工具。搭建整个解决方案框架的步骤如下：

（1）创建一个空白解决方案，由于解决方案中项目较多，为了使项目结构更加清晰，在解决方案中添加两个解决方案文件夹：DLL 和 Test，用以组织各个项目。

（2）在解决方案 DLL 文件夹中，添加一个类库项目 SJL.Entity 作为实体框架层。

（3）在解决方案 DLL 文件夹中，添加一个类库项目 SJL.Dal 作为数据访问层。

（4）在解决方案 DLL 文件夹中，添加一个类库项目 SJL.Bll 作为业务逻辑层。

（5）在解决方案 DLL 文件夹中，添加一个类库项目 SJL.Common，此项目中包含各个项目中用到的公共类。

（6）在解决方案中添加一个 ASP.NET Web 应用程序项目 SJL.Web，作为表现层。

（7）在解决方案 Test 文件夹中添加一个类库项目 DalTest，作为数据访问层的单元测试项目。

（8）在解决方案 Test 文件夹中添加一个类库项目 BllTest，作为业务逻辑层的单元测试项目。

（9）在各个项目之间添加引用关系，表现层、业务逻辑层、数据访问层都引用实体框架层和公共类库，业务逻辑层引用数据访问层，表现层引用业务逻辑层，测试项目引用被测试项目。整个解决方案结构如图 12.2 所示。

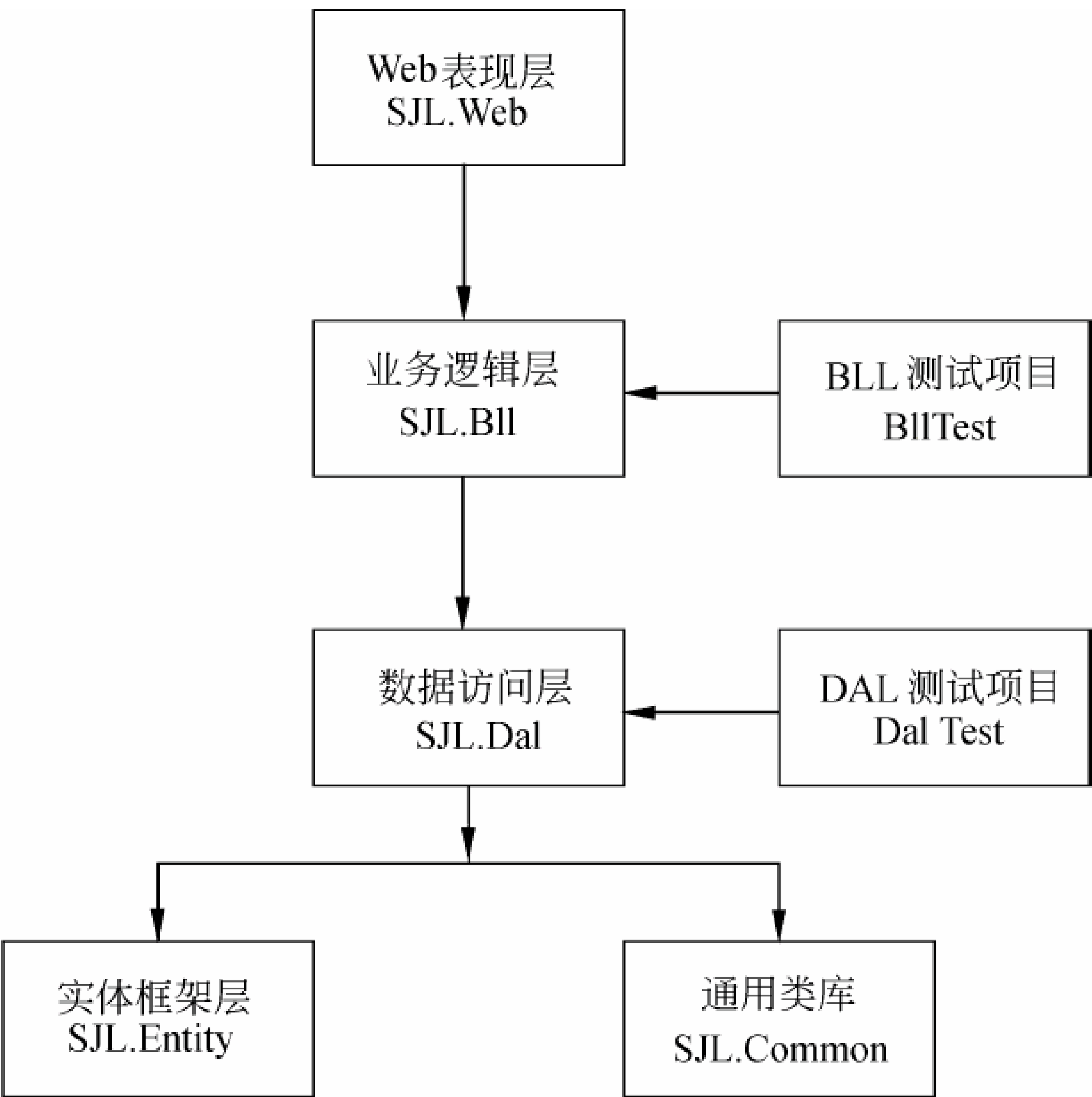


图 12.2　解决方案结构

实体框架层 SJL.Entity 的实体框架结构如图 12.3 所示。

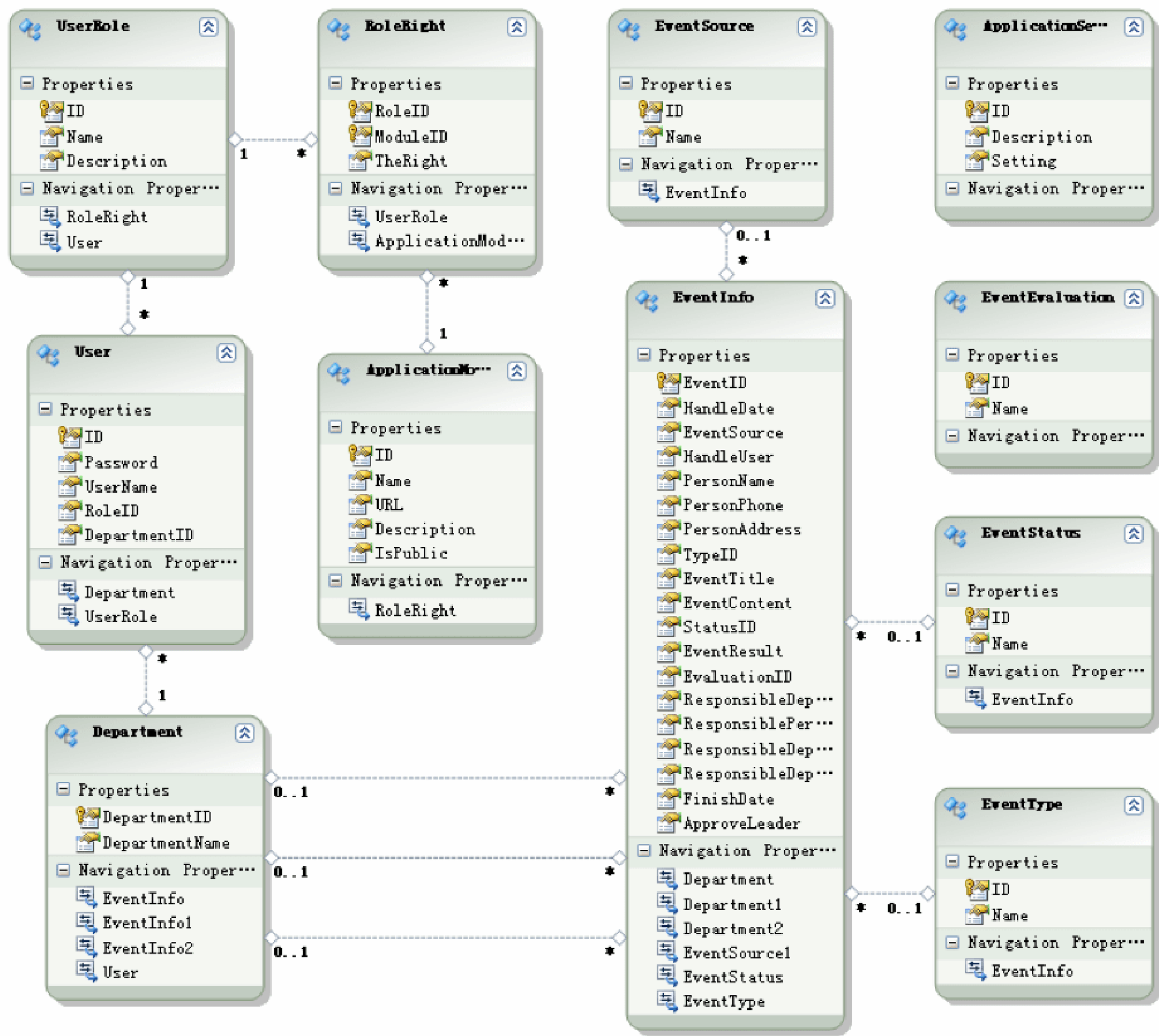


图 12.3　实体框架结构

12.2　主题和母版页

一个正规的 Web 应用程序应该具有整齐、规范、一致的界面风格。ASP.NET 的主题和母版页技术为维护网站外观提供了有效的手段，合理地应用主题和母版，能够用较少量的代码设计出美观的页面。本节将介绍县长公开电话受理系统项目中的主题和母版页设计。

12.2.1　主题设计

本项目外观设计风格采用浅蓝色为主色调，配合使用浅灰、浅黄等颜色，不使用非常鲜艳的颜色，避免出现强烈的视觉反差，使设计出的页面干净清新，稳重而不呆板，美观而不花哨。本项目对最经常使用的控件包括 GridView、Button、TextBox、DropDownList，设计了默认主题和外观，具体实现步骤如下：

（1）在表现层项目 SJL.Web 中添加一个名称为 default 的主题文件夹。

（2）在 default 主题文件夹中添加一个外观文件 CommonControl.skin。

（3）在 CommonControl.skin 文件中编写代码，为服务器端控件 GridView 和 Button 指定外观，其中 Button 控件应用了 jQuery UI 中 Button 的样式。

提示：jQuery UI 用到的 CSS 文件可以从 http://jqueryui.com/download 下载。

```
<asp:GridView  runat="server"  CellPadding="4"    ForeColor="#333333"
GridLines="None">
       <RowStyle CssClass="oddRow" />
       <FooterStyle BackColor="#507CD1" Font-Bold="True" ForeColor="White"
       />
       <PagerStyle BackColor="#2461BF" ForeColor="White" HorizontalAlign=
       "Center" />
       <SelectedRowStyle BackColor="#D1DDF1" Font-Bold="True" ForeColor=
       "#333333" />
       <HeaderStyle BackColor="#507CD1" Font-Bold="True" ForeColor="White"
       />
       <EditRowStyle BackColor="#2461BF" />
       <AlternatingRowStyle  CssClass="evenRow" />
       <EmptyDataTemplate> <div style="color:red; font-size:14px; border:
       solid 1px blue; padding:30px; ">没有符合条件的数据，请添加数据或修改查询
       条件。</div>  </EmptyDataTemplate>
</asp:GridView>
<asp:Button runat="server" CssClass="ui-button ui-state-default ui-corner-
all"></asp:Button>
```

（4）在 default 文件夹里添加一个 CSS 文件，在其中定义主题中用到的样式和常用 HTML 控件样式。

```
.oddRow{ background-color:#EFF3FB;}                /*GridView 奇数行背景色*/
.evenRow{background-color:White;}                  /*GridView 偶数行背景色*/
.hoverRow{background-color:#ffffdd;}               /*GridView 鼠标划过背景色*/
input[type=text]{ border:solid 1px #9af;}          /*文本框样式*/
input[type=text][readonly]{background-color:#eee;} /*只读文件框样式*/
select{border:solid 1px; background-color:#ffd; width:150px;}
                                                   /*下拉表样式*/
```

上述主题样式的外观如图 12.4 所示。

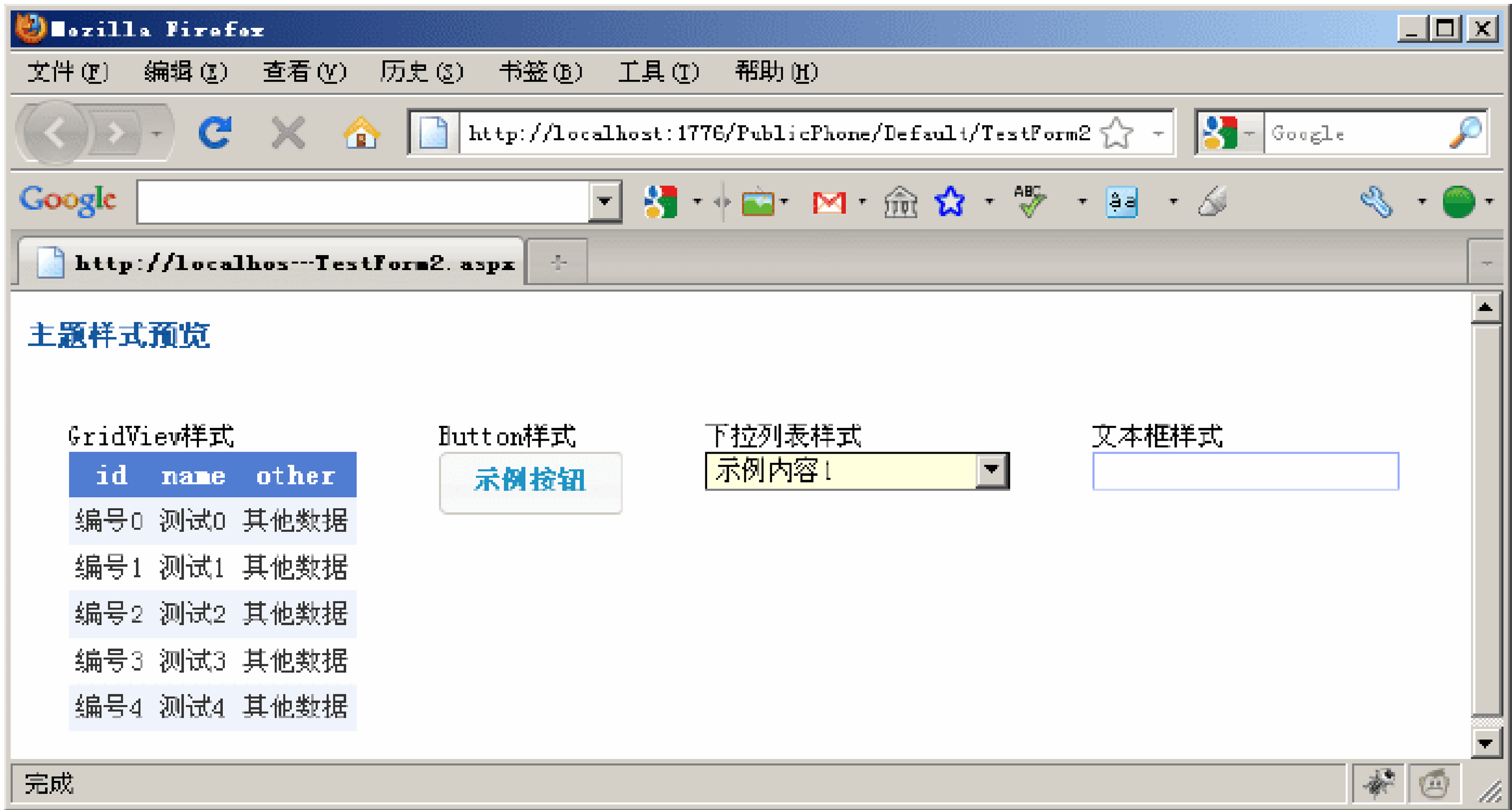


图 12.4　主题样式预览

12.2.2　母版页设计

公开电话受理系统中页面布局分为 3 部分：页头、正文和页脚，其中页头显示工具栏，

页脚显示网站说明和联系方式，这两部分内容对于大多数页面来说是相同的，属于母版页的内容。公开电话受理系统母版页的实现步骤如下：

（1）在表现层项目 SJL.Web 中添加一个文件夹 UserControls，用于存放项目中的用户控件。

（2）在 UserControls 文件夹中添加一个用户控件 Header.ascx，用于实现页面头部，其中主要包含简单的提示信息和功能导航栏。Header.ascx 用户控件设计外观如图 12.5 所示。

图 12.5　Header 用户控件

Header.ascx 用户控件代码如下：

```
<%--这个 div 显示程序名称、当前日期和当前用户--%>
<div style="background:#f2f2fe; width:1000px; text-align:left;">
<div>
 <span style="color:#005599; font-size:24px; font-weight:bold; font-family:
 华文楷体;" >
  A 县书记、县长公开电话受理平台</span>
 <span style="margin-left:120px; margin-right:30px;">
 当前日期：
 <%=DateTime.Now.ToLongDateString()+" "+DateTime.Today.DayOfWeek %>
 当前用户：
 <%= SJL.Web.HttpCode.WebUtility.currentUser == null ? "未知用户" :
SJL.Web.HttpCode.WebUtility.currentUser.UserName%>
   【<a href="../LogoutPage.aspx">退出登录</a>】 </span>
</div>
</div>
<%--下面这个 div 为功能导航栏--%>
<div id="NavMenu">
<div class="ImageMenu"><img src="../images/policy.png" alt="电话业务受理"
/><br/>
<a href="../Default/EventEditPage.aspx?id=-1">电话业务受理</a></div>
<div class="ImageMenu"><img src="../images/report.png" alt="业务详情查询"
/><br/>
<a href="../Default/EventListPage.aspx">受理业务查询</a></div>
<div class="ImageMenu"><img src="../images/user.png" alt="用户权限管理"
/><br/>
<a href="../UserRight/ChangePassPage.aspx">用户权限管理</a></div>
<div class="ImageMenu"><img src="../images/dictionary.png" alt="数据字典管
理" /><br/>
<a href="../Dictionary/DepartmentPage.aspx">数据字典管理</a></div>
</div>
```

在 Header.ascx 用户控件中，用 CSS 样式控制导航菜单向左浮动以及文字和图片的上下对齐方式，CSS 代码如下：

```
/*div#NavMenu 页面顶部导航菜单*/
div#NavMenu{background-color: #003399; font-size:medium;
font-weight:bold; height:70px; vertical-align:middle;}
div#NavMenu a {color:White; }
div#NavMenu a:hover{ background-color:#ddddff; color:black;    }
/*带图片的菜单，上图片，下文字*/
div#NavMenu  div.ImageMenu{  text-align:center;  float:left;  margin:auto
10px;}
div#NavMenu div.ImageMenu img{height:48px;}
```

（3）在 UserControls 文件夹中添加一个用户控件 Footer.ascx，用于实现网站页脚。

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="Footer.ascx.
cs" Inherits="SJL.Web.UserControls.Footer" %>
<div style="width:1000px; text-align:center;
   border:dashed 1px silver; background:#f2f2f2; float:none; clear:both;">
   A 县书记、县长公开电话受理平台。公开电话：(0543)5350686；5326899。<br />
主 办 单 位 ： 滨 州 市  A  县 人 民 政 府 。 技 术 支 持 ：  <a  href=
"mailto:sun.j.l.studio@gmail.com"> sun.j.l.studio@gmail.com </a>
</div>
```

（4）在 SJL.Web 项目中添加一个母版页 SjlMaster.Master，在母版页中放置 3 个 div，在第 1 个 div 中放置 Header 用户控件，第 2 个 div 中放置一个 ContentPlaceHolder，第 3 个 div 中放置 Footer 用户控件，代码如下：

```
<div id="main">
<form id="form1" runat="server">
<div>  <uc1:Header ID="Header1" runat="server" />  </div>
<div>
<asp:ContentPlaceHolder ID="content" runat="server">  </asp:ContentPlace-
Holder>
</div>
<div>  <uc2:Footer ID="Footer1" runat="server" />  </div>
</form>  </div>
```

为了使整个页面在不同分辨率下保持固定大小，并且在浏览器中居中显示，需要对页面最外层的 div 进行样式设置，设定其宽度、水平对齐方式，CSS 代码如下：

```
/*最外层 DIV，整个页面居中*/
div#main{width:1000px;  margin-left:auto;  margin-right:auto;  }
```

（5）在母版页 SjlMaster.master 中创建一个内容页面并测试母版页外观，运行界面如图 12.6 所示。



图 12.6　母版页外观

12.3　电话业务受理

当有群众通过县长公开电话反映问题时，公开电话受理中心的工作人员需要将问题的详情通过程序记录下来。本节将讨论这一功能的设计和实现。

12.3.1　事件编号生成算法

在县长公开电话受理系统项目中，群众通过电话所反映的一个问题称为一个事件，事件信息保存在数据库的 EventInfo 表中。在添加新事件时，需要为新事件生成一个唯一的事件编号，从数据库的角度来说，要为这条新记录确定一个主键。生成事件编号最简单的办法就是使用数据库的标识列（自动增长列），但是这样生成的主键可读性很差。在实际应用中，通用使用具有一定字面含义的文本作为主键，例如，可能用某种算法基于当前日期生成一个字符串作为主键。

在公开电话受理项目中，使用当前日期（yymmdd 格式的 6 位数字）加 4 位顺序号组合为 10 位数字作为主键，例如 0910030212、1002050018 等。每当添加一个新的事件信息时，都需要根据此算法为新事件生成一个主键。为了使顺序号递增，需要记录上一个事件的顺序号，将上一事件顺序号加 1 即可得到当前事件顺序号，当顺序号增加到 9999 时恢复为 0。为了提高查询速度和简化顺序号生成算法，在数据库中使用一个表 ApplicationSetting 来保存当前顺序号。根据以上分析，新增事件编号生成算法如下：

（1）取得当前日期，生成 yymmdd 格式的 6 位数字。

（2）从数据库 ApplicationSetting 表中取得当前顺序号，转换为 4 位数字，不足 4 位数字的左侧用 0 补齐。

（3）将数据库 ApplicationSetting 表中当前顺序号加 1 并保存，如果顺序号加 1 超过 9999（4 位整数能表示的最大值），则将顺序号重置为 0 并保存到数据库。

（4）将 6 位日期与 4 位顺序号组合，即得到新事件编号。

事件编号生成算法对应代码如下（位于数据访问层项目的 EventInfoDAL 类中）：

```
public static string getNewEventID()
{
    var context = new UserRightContext();   //创建实体容器
    int n=context.getPartialNewEventID();   //调用存储过程得到事件顺序号
    string s=DateTime.Now.ToString("yyMMdd") + n.ToString("0000");
                                            //得到完整事件编号
    context.Dispose();
    return s;
}
```

上述代码中用到的存储过程 sp_GetPatialNewEventID 代码如下：

```
CREATE PROCEDURE [dbo].[sp_GetPartialNewEventID]
AS
BEGIN
    set nocount on
    declare @ret int
```

```
    begin transaction
    select @ret=cast(Setting as int) from ApplicationSetting  where
    ID=1;
    if @ret>=9999 set @ret=1;
    update ApplicationSetting set Setting=@ret+1 where ID=1;
    commit transaction
    select @ret
END
```

12.3.2　数据访问层和业务逻辑层

公开电话受理系统是一个多层结构的 Web 应用程序，与数据库相关的代码都在数据访问层中实现。为了实现电话业务受理功能，需要在数据访问层中实现相应的数据操作功能，主要包括事件的添加、修改和查询。

（1）在数据访问层项目 SJL.Dal 中添加一个类 EventInfoDal。

```
public static class EventInfoDAL
{   }
```

（2）为提高数据访问层代码的可靠性，尽早发现程序中的 bug，应该在第 1 时间对数据访问层代码进行单元测试。在测试项目 DalTest 中添加一个新的单元测试类 EventInfoDalTest，此类将测试 EventInfoDal 中的方法。

```
[TestClass]
public class EventInfoDalTest
{   }
```

（3）在 EventInfoDal 类中添加一个 getByID()方法，实现根据事件 ID 取得事件详情的功能。

```
public static EventInfo getByID(string id)     //根据事件 ID 得到事件信息
{
   var query = getQuery();
   var target= EntityUtility.selectOne(query,e=>e.EventID==id,false,
   false);
   loadReference(target);
   EntityUtility.detachEntity(query.Context, target);
   query.Context.Dispose();
   return target;
}
```

（4）在 EventInfoDalTest 类中添加一个方法，对 EventInfoDAL.getByID()方法进行测试。

```
   [TestMethod]
   public void testGetByID()                     //根据事件 ID 得到事件信息
   {
      string id = "0018";                        //一个已经存在的事件 ID
      var e = EventInfoDAL.getByID(id);
      Assert.IsNotNull(e);                       //所返回的事件不为空
      Assert.IsNotNull(e.HandleUser);            //事件受理者不为空
      Assert.AreEqual(id, e.EventID);            //事件 ID 正确
   }
```

（5）运行 EventInfoDalTest.testGetById 测试方法，如有错误则及时修改，测试通过后

进行下一步。

（6）在 EventInfoDal 中添加一个 add()方法，实现添加事件的功能。

```
//添加事件
public static int add(EventInfo info)
{
    return EntityUtility.add<UserRightContext,EventInfo>(info);
}
```

（7）在 EventInfoDalTest 中添加一个方法，对 EventInfoDAL.add()方法进行测试。

```
[TestMethod]
public void testAdd()
{
    //创建一个新事件并设置其主要属性
    EventInfo e = new EventInfo();
    string id=EventInfoDAL.getNewEventID();
    e.EventID = id;
    e.EventTitle = "test";
    e.HandleUser = "admin";
    e.HandleDate = DateTime.Now;
    EventInfoDAL.add(e);                          //添加事件
    var  e2 = EventInfoDAL.getByID(id);           //获取添加的事件
    Assert.IsNotNull(e2);
}
```

（8）运行 EventInfoDALTest.testAdd 测试方法，如有错误则及时修改，测试通过后进行下一步。

（9）在 EventInfoDal 类中添加一个 update()方法，实现修改事件的功能。

```
//修改事件
public static int update(EventInfo info)
{
    return EntityUtility.update<UserRightContext, EventInfo>(info, "Event
    ID");
}
```

（10）在 EventInfoDalTest 类中添加一个方法，对 EventInfoDAL.update()方法进行测试。

```
[TestMethod]
//修改事件
public void testUpdate()
{
    string id = "0018";                           //一个已经存在的事件 ID
    var e = EventInfoDAL.getByID(id);             //首先取得事件
    //以下语句修改事件各个属性
    string department = "02";
    int evaluation = 3;
    string person = "zx02";
    e.EvaluationID = evaluation;
    e.ResponsibleDepartment = department;
    e.ResponsiblePerson = person;
    EventInfoDAL.update(e);                       //将事件更新到数据库
    e = EventInfoDAL.getByID(id);                 //重新从数据库取得事件
    //以下语句测试所取得的事件信息已经变成修改后的值
    Assert.AreEqual(department, e.ResponsibleDepartment);
    Assert.AreEqual(person, e.ResponsiblePerson);
    Assert.AreEqual(evaluation, e.EvaluationID);
}
```

（11）在 EventInfoDal 类中添加一个 delete()方法，实现删除事件功能。

```
//删除事件
public static int delete(string id)
{
    return EntityUtility.delete(getQuery(),e=>e.EventID==id);
}
```

（12）在 EventInfoDalTest 类中添加一个方法，对 EventInfoDAL.delete()方法进行测试。

```
[TestMethod]
public void testDelete()
{
    //为避免删除数据库中原有数据，先添加一个新事件然后再删除
    EventInfo e = new EventInfo();
    string id = EventInfoDAL.getNewEventID();
    e.EventID = id;
    e.EventTitle = "test";
    e.HandleUser = "admin";
    e.HandleDate = DateTime.Now;
    EventInfoDAL.add(e);                        //添加事件
    var e2 = EventInfoDAL.getByID(id);          //获取添加的事件
    Assert.IsNotNull(e2);
    EventInfoDAL.delete(id);
    e2 = EventInfoDAL.getByID(id);              //执行删除
    Assert.IsNull(e2);                          //删除后数据库中不存在此事件
}
```

（13）业务逻辑层的功能简单，只需要调用相应的数据访问层方法即可。在业务逻辑层项目 SJL.Bll 中添加一个类 EventInfoBLL，在类中添加增、删、改、查功能。

```
public static class EventInfoBLL
{
    private const string NewEventID = "";
    [DataObjectMethod(DataObjectMethodType.Insert)]
    public static int add(EventInfo info)
    {
        if (string.IsNullOrEmpty(info.EventID))
            throw new ArgumentNullException("EventID");
        return DB.add(info);
    }
    public static int update(EventInfo info)
    {
        return DB.update(info);
    }
    public static int delete(string id)
    {
        return DB.delete(id);
    }
    public static EventInfo getByID(string id)
    {
        return DB.getByID(id);
    }
    public static EventInfo newEventInfoWithID()
    {
        return new EventInfo() { EventID = NewEventID};
    }
    public static string getNewEventID()
```

```
    {
        return DB.getNewEventID();
    }
```

12.3.3　事件详情用户控件

在公开电话受理系统项目中，会在多个页面中显示、修改、添加事件信息。为了避免重复代码，遵循一个功能只实现一次的原则，应该将事件详情显示和编辑界面封装成一个用户控件，再放到不同的页面中使用。创建事件详情用户控件步骤如下：

（1）在表现层项目 SJL.Web 中添加一个文件夹 UserControls，用来存放项目中的所有 Web 用户控件。在 UserControls 文件夹事件中新建一个用户控件 EventInfoControl.ascx，并参照图 12.7 对页面进行设计。

图 12.7　事件详情用户控件

在图 12.7 所示的事件详情用户控件 EventInfoControl.ascx 中，整个页面使用 table 布局，受理日期和办结日期都启用了 jQuery 日历控件，反映内容和答复意见分别是一个多行的 TextBox 控件，用一个隐藏域 HiddenField 保存当前正在编辑的事件编号。EventInfoControl.ascx 代码如下：

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="EventInfo
Control.ascx.cs" Inherits="SJL.Web.UserControls.EventInfoControl" %>
<script type="text/javascript">
    $(function() {
    $('#<%=theDate.ClientID%>').datepicker();  //启用 jQuery 日历控件
    $('#<%=finishDate.ClientID%>').datepicker();
    });
 </script>
<table style="background:#eff; ">
<tr>
<td colspan="8">
    <asp:HiddenField ID="hiddenOldID" runat="server" />
    </td>
</tr>
<tr><td>事项编号</td>
```

```
<td><asp:TextBox runat="server" ID="eventId" Text=""></asp:TextBox></td>
<td>受理日期</td>
<td><asp:TextBox ID="theDate" runat="server" Text="" ></asp:TextBox></td>
<td>受理类别</td>
<td> <asp:DropDownList ID="eventSourceList" runat="server"> </asp:Drop-
DownList></td>
<td>受理人员</td>
<td> <asp:TextBox ID="handleUser" runat="server"></asp:TextBox></td></tr>
<tr><td >反映人</td>
<td> <asp:TextBox ID="personName" runat="server"></asp:TextBox> </td>
<td>电话</td>
<td> <asp:TextBox ID="personPhone" runat="server"></asp:TextBox></td>
<td>地址</td><td> <asp:TextBox ID="personAddress" runat="server"></asp:
TextBox> </td>
<td>满意程度</td><td><asp:DropDownList runat="server" ID="evaluationList"
/></td></tr>
<tr><td>承办单位</td>
<td><asp:DropDownList ID="departmentList" runat="server" DataValueField=
"DepartmentID" DataTextField="DepartmentName"> </asp:DropDownList></td>
<td>承办人员</td>
<td>  <asp:TextBox  ID="responsiblePerson"  runat="server"></asp:TextBox>
</td>
<td>承办单位 2</td>
<td><asp:DropDownList ID="departmentList2" runat="server" DataValueField=
"DepartmentID" DataTextField="DepartmentName"> </asp:DropDownList></td>
<td>承办单位 3</td>
<td><asp:DropDownList ID="departmentList3" runat="server" DataValueField=
"DepartmentID"  DataTextField="DepartmentName">  </asp:DropDownList></td>
</tr>
<tr> <td>问题类型</td>
<td><asp:DropDownList ID="eventTypeList" runat="server"></asp:DropDown-
List></td>
<td>处理状态</td>
<td><asp:DropDownList ID="eventStatusList" runat="server"> </asp:DropDown
List></td>
<td>办结日期</td>
<td> <asp:TextBox ID="finishDate" runat="server" Text="" ></asp:TextBox>
</td>
<td>签批领导</td>
<td> <asp:TextBox ID="leader" runat="server"></asp:TextBox> </td> </tr>
<tr><td>反映主题</td>
<td colspan="7"><asp:TextBox ID="eventTitle" runat="server" Width="800">
</asp: TextBox></td></tr>
<tr><td>反映内容</td>
<td colspan="7"> <asp:TextBox ID="eventContent" runat="server" Height=
"200px" Width="800px"
 TextMode="MultiLine"> </asp:TextBox>  </td> </tr>
<tr><td>答复意见</td>
<td    colspan="7">    <asp:TextBox    ID="eventResult"    runat="server"
Height="120px" Width="800px"
 TextMode="MultiLine"></asp:TextBox> </td> </tr>
</table>
```

（2）在用户控件 EventInfoControl.ascx 中有多个下拉列表 DropDownList 控件，包括部门列表、问题类型列表、处理状态列表、满意程度列表。这些下拉列表控件都需要绑定到数据库中的数据，一般用如下代码实现（以问题类型列表为例）：

```
var list=EventTypeBll.getAll();           //从数据库中得到问题类型数据
eventTypeList.DataValueField="ID";        //设置值字段
eventTypeList.DataTextField="Name";       //设置显示字段
eventTypeList.DataSource=list;
eventTypeList.DataBind();
```

用户控件 EventInfoControl.ascx 中有 4 个下拉表，如果为每个下拉表都写同样代码，则重复代码太多，而且其他页面中也存在类似的很多下拉表。利用泛型和接口可以编写一个通用方法实现数据绑定功能，如下代码所示：

```
//假设数据字典只有两个属性（字段）：ID 和 Name
//数据字典业务逻辑层只有一个方法：返回所有数据
public interface IDictionaryBll<T>            //数据字典业务逻辑层接口
{
    List<T> getAll();
}
public class EventEvaluationBll:IDictionaryBll<EventEvaluation>
                                              //事件评价业务逻辑层
{
    public List<EventEvaluation> getAll()
    {
        return EventEvaluationDal.getAll();
    }
}
public class EventSourceBll:IDictionaryBll<EventSource>
                                              //事件来源业务逻辑层
{
    public List<EventSource> getAll()
    {
        return EventSourceDal.getAll();
    }
}
public class EventStatusBll:IDictionaryBll<EventStatus>
                                              //事件处理状态业务逻辑层
{
    public List<EventStatus> getAll()
    {
        return EventStatusDal.getAll();
    }
}
public class EventTypeBll:IDictionaryBll<EventType>//事件类型业务逻辑层
{
    public List<EventType> getAll()
    {
        return EventTypeDal.getAll();
    }
}
//Web 工具类，包含几个常用方法
public static class WebUtility
{
    ///<summary>
    ///将数据字典绑定到下拉列表控件中
    ///</summary>
    ///<typeparam name="TBll">获取数据字典数据的业务逻辑层类型</typeparam>
    ///<typeparam name="TElement">业务实体类型</typeparam>
    ///<param name="dropdown">要绑定的下拉列表控件</param>
   public static void bindListWithDictionary<TBll, TElement>(DropDownList
   list)
```

```
        where TBll : IDictionaryBll<TElement>,new()
    {
        var l= new TBll().getAll();          //调用业务逻辑层方法返回数据
        list.DataValueField = "id";
        list.DataTextField = "name";
        list.DataSource = l;
        list.DataBind();
    }
}
```

（3）在用户控件 EventInfoControl.ascx 的 Page_Load 事件中，利用上一步所实现的通用数据绑定方法将数据绑定到下拉列表中：

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
        initList();
}
//下拉列表是否已经绑定数据（防止重复绑定）
private bool listInitialized
{
    get { return WebUtility.convertToBoolean(ViewState [ViewStateInit-
    List]); }
    set { ViewState[ViewStateInitList] = value; }
}
//为下拉列表填充数据
private void initList()
{
    if (listInitialized) return;
    //下面 4 条语句调用泛型方法绑定数据列表
    WebUtility.bindListWithDictionary<EventSourceBll, EventSource> (event
    SourceList);
    WebUtility.bindListWithDictionary<EventTypeBll, EventType> (eventTyp-
    eList);
    WebUtility.bindListWithDictionary<EventStatusBll, EventStatus>(event-
    StatusList);
    WebUtility.bindListWithDictionary<EventEvaluationBll, EventEvaluati-
    on> (evaluationList);
    initDepartmentList();
    listInitialized = true;
}
// 为部门列表填充数据
private void initDepartmentList()
{
    if (listInitialized) return;
    var departments=DepartmentBLL.getAll(PageDataArgument.allData);
    DropDownList[] lists={departmentList,departmentList2,departmentL-
    ist3};
    for (int i = 0; i < 3; i++)          //将部门列表绑定到 3 个下拉列表中
     {
        lists[i].DataSource= departments;
        lists[i].DataBind();
        //在部门列表的第 1 项插入"请选择"字样
        lists[i].Items.Insert(0, new ListItem("--请选择--", "-1"));
        lists[i].SelectedIndex = 0;
     }
}
```

（4）用户控件 EventInfoControl.ascx 用于显示和编辑事件详情，当保存事件信息时，

需要区分当前被编辑的事件是否为空，是新增事件，还是原有事件。可以在用户控件中添加属性以区分事件数据的不同状态：

```
public partial class EventInfoControl : System.Web.UI.UserControl
{
#region constants
//以下常量皆为 ViewState 中的键值
private const string ViewStateIsNewRecord = "isNewRecord";
private const string ViewStateIsNullRecord = "isNullRecord";
private const string ViewStateInitList = "initlist";
private const string ViewStateTheEvent = "theEvent";
#endregion
//用户控件当前显示的是否一条新记录
private bool isNewRecord
{
    get
    {
        return WebUtility.convertToBoolean(ViewState[ViewStateIsNewRec-
        ord]);
    }
    set
    {
        ViewState[ViewStateIsNewRecord] = value;
    }
}
//用户控件当前显示的是否一条空记录
private bool isNull
{
    get
    {
        return WebUtility.convertToBoolean(ViewState[ViewStateIsNullRec-
        ord]);
    }
    set
    {
        ViewState[ViewStateIsNullRecord] = value;
    }
}
}
```

（5）在用户控件 EventInfoControl.ascx 中添加一个 EventInfo 类型的属性，实现用户界面和实体类之间的双向数据传递：

```
// 与此用户控件相对应的 EventInfo
public EventInfo theEvent
{
    get { return getEventInfo(); }
    set { setEventInfo(value); }
}
/// <summary>
/// 根据用户控件生成事件信息（一个 EventInfo 类的实例）
/// </summary>
/// <returns>得到的事件信息</returns>
private EventInfo getEventInfo()
{
    if (isNull) return null;                    //如果为空记录则返回 null
    if (!listInitialized)
        initList();
    if (ViewState[ViewStateTheEvent] == null)   //将事件信息保存在 ViewState
```

```
        ViewState[ViewStateTheEvent] = new EventInfo();
    EventInfo info = ViewState[ViewStateTheEvent] as EventInfo;
    //以下语句根据页面上各控件的值设置 EventInfo 各个属性
    info.EventID = eventId.Text;
    info.HandleDate = DateTime.Parse(theDate.Text) ;
    info.HandleUser = handleUser.Text;
    info.EventSource = int.Parse(eventSourceList.SelectedValue);
    info.PersonName = personName.Text;
    info.TypeID = int.Parse(eventTypeList.SelectedValue);
    info.EventContent = eventContent.Text;
    info.StatusID = int.Parse(eventStatusList.SelectedValue);
    string department;
    department=departmentList.SelectedValue;
    info.ResponsibleDepartment = department == "-1" ? "" : department;
    info.ResponsiblePerson = responsiblePerson.Text;
    department = departmentList2.SelectedValue;
    info.ResponsibleDepartment2 = department == "-1" ? null : department;
    department = departmentList3.SelectedValue;
    info.ResponsibleDepartment3 = department == "-1" ? null : department;
    info.EvaluationID = int.Parse(evaluationList.SelectedValue);
    info.PersonAddress = personAddress.Text;
    info.PersonPhone = personPhone.Text;
    info.EventResult = eventResult.Text;
    info.EventTitle = eventTitle.Text;
    info.ApproveLeader = leader.Text;
    DateTime t;
    if (DateTime.TryParse(finishDate.Text, out t)) //判断是否输入合法日期
        info.FinishDate = t;
    else
        info.FinishDate = null;
    return info;
}
/// <summary>
/// 根据事件信息设置用户控件
/// </summary>
/// <param name="info">要设置的事件信息</param>
private void setEventInfo(EventInfo info)
{
    if (!listInitialized)
        initList();
    isNewRecord = false;
    ViewState[ViewStateTheEvent] = info;
    if (info == null)                               //如果事件为空则清空界面
    {
        isNull = true;
        hiddenOldID.Value = "";
        WebUtility.clearFormInput(this);
        return;
    }
    hiddenOldID.Value = info.EventID;               //保存原事件 ID
    isNull = false;
    //以下语句根据 EventInfo 的各个属性设置相应控件内容
    eventId.Text = info.EventID;
    theDate.Text = info.HandleDate.Value.ToShortDateString();
    handleUser.Text = info.HandleUser;
    //根据指定的值选中下拉列表中某一项
    WebUtility.selectByValue(eventSourceList,info.EventSource.To-
    String());
    personName.Text = info.PersonName;
    WebUtility.selectByValue(eventTypeList,info.TypeID.ToString());
```

```
    eventContent.Text = info.EventContent;
    WebUtility.selectByValue(eventStatusList, info.StatusID.ToString());
    WebUtility.selectByValue(departmentList,info.ResponsibleDepartment);
    WebUtility.selectByValue(departmentList2, info.ResponsibleDepartm-
    ent2);
    WebUtility.selectByValue(departmentList3, info.ResponsibleDepartm-
    ent3);
    WebUtility.selectByValue(evaluationList, info.EvaluationID.ToStr-
    ing());
    responsiblePerson.Text = info.ResponsiblePerson;
    personPhone.Text = info.PersonPhone;
    personAddress.Text = info.PersonAddress;
    eventResult.Text = info.EventResult;
    eventTitle.Text = info.EventTitle;
    if (info.FinishDate.HasValue)
        finishDate.Text = info.FinishDate.Value.ToShortDateString();
    else
        finishDate.Text = "";
    leader.Text = info.ApproveLeader;
}
```

上述代码用到了一个 WebUtility 类中的一个静态方法 selectByValue()，这个方法的作用是根据指定的值从下拉列表控件中选中匹配项，如下代码所示：

```
/// <summary>
/// 从下拉列表中选中具有指定值的一项
/// </summary>
/// <param name="list">下拉列表控件</param>
/// <param name="value">要选定的值</param>
/// <returns>所选中的项的索引，如果不存在对应项则返回-1</returns>
public static int selectByValue(DropDownList list,string value)
{
    if(list.Items.Count==0)
        return -1;
    if (!string.IsNullOrEmpty(value))
    {
        for (int i = 0; i < list.Items.Count; i++)
        {
            if (list.Items[i].Value == value)        //找到指定值
            {
                list.SelectedIndex = i;
                return i;
            }
        }
    }
    list.SelectedIndex = 0;
    return -1;
}
```

（6）在用户控件 EventInfoControl.ascx 中添加一个方法 setByID()，实现根据事件编号填充控件内容的功能：

```
/// <summary>
/// 根据事件编号设置用户控件内容
/// </summary>
/// <param name="id">事件编号</param>
public void setByID(string id)
{
    if (string.IsNullOrEmpty(id))                //如果编号为空则清空控件
```

```
        setEventInfo(null);
    else
        setEventInfo(EventInfoBLL.getByID(id));
                                    //否则调用业务逻辑层得到事件信息并设置控件内容
}
```

（7）在用户控件 EventInfoControl.ascx 中添加一个 save()方法，用以保存事件数据。在保存时需要判断此事件是新增事件还是原有事件，从而分别调用业务逻辑层的 add 或者 update()方法实现添加或更新。在更新原有数据时，还要判断主键事件编号是否发生改变，如果发生了改变，则需要先将原有事件删除，再添加新的事件：

```
/// <summary>
/// 保存（插入或更新）事件信息
/// </summary>
public int save()
{
    var e = theEvent;
    if (e == null) return 0;
    int n = 0;
    if (isNewRecord)                                        //新添加的事件
    {
        n = EventInfoBLL.add(e);
        eventId.Text = e.EventID;
    }
    else                                                    //原有事件
    {
        if (keyChanged)                                     //主键发生了改变
        {
            n = EventInfoBLL.add(e);                        //插入新记录
            n = EventInfoBLL.delete(hiddenOldID.Value);     //删除旧记录
        }
        else
            n = EventInfoBLL.update(e);
    }
    isNewRecord = false;
    return n;
}
/// <summary>
/// 事件编号是否发生了改变
/// </summary>
private bool keyChanged
{
    get
    {
        if (isNewRecord) return true;
        //通过比较 HiddenField 保存的原有事件编号与 TextBox 控件中的事件编号确定是否
        //改变
        return hiddenOldID.Value != eventId.Text.Trim();
    }
}
```

12.3.4　电话业务受理页面

电话业务受理页面允许用户输入群众电话所反映事件的详细信息并保存到数据库。此页面是在 12.3.3 节所介绍的事件详情用户控件 EventInfo.ascx 基础上实现的，由于

EventInfo.ascx 控件实现了事件编辑和保存的功能，因此电话业务受理页面代码较少。创建电话业务受理页面具体操作步骤如下：

（1）在表现层项目 SJL.Web 的 UserControls 文件夹中添加一个事件编辑用户控件 EventEdit.ascx，此控件在事件详情用户控件 EventInfo.ascx 基础上添加了保存、取消等按钮，代码如下：

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="EventEdit
Control.ascx.cs" Inherits="SJL.Web.UserControls.EventEditControl" %>
<%@  Register  src="EventInfoControl.ascx"  tagname="EventInfoControl"
tagprefix="uc1" %>
<asp:Button  ID="ok"  runat="server"  Text=" 保 存 "  onclick="ok Click"
/>  
<input type="button" onclick="window.location=window.location;" value="
取消" />  
<input type="button" onclick="window.location='EventEditPage.aspx?id=
-1';" value="新建" />  
<input type="button" onclick="window.close();" value="关闭" />
<!-- <input type="button" onclick="alert('此功能尚未实现。');" value="流转"
/>  -->
<uc1:EventInfoControl ID="eventInfo1" runat="server" />
```

（2）为 EventEdit.ascx 控件的几个按钮编写代码：

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        //QueryString 中 ID 参数表示要编辑的事件编号
        string s = Request.QueryString["id"];
        if (s == "-1")                //如果为-1 则是添加新事件
            eventInfo1.newRecord();
        else
            eventInfo1.setByID(s);
    }
}
protected void cancel Click(object sender, EventArgs e)
{
    eventInfo1.theEvent = null;      //单击取消按钮，清空控件内容
}
protected void ok Click(object sender, EventArgs e)
{
    //单击确定按钮，保存事件信息并提示
    int n=eventInfo1.save();
    if(n>0)
        Page.ClientScript.RegisterStartupScript(this.GetType(), "succe-
        ss", "alert('保存成功！');", true);
}
protected void add Click(object sender, EventArgs e)
{
    eventInfo1.newRecord();          //单击添加按钮时产生一条新记录
    ok.Enabled = true;
}
```

（3）在 SJL.Web 项目中添加一个新页面 EventEditPage.aspx，在页面中放置一个上一步所创建的 EventEdit.ascx 控件：

```
<asp:Content ID="Content2" ContentPlaceHolderID="content" runat="server">
```

```
 <h3>电话业务受理</h3>
 <uc1:EventEditControl ID="EventEditControl1" runat="server" />
</asp:Content>
```

（4）运行 EventEditPage.aspx，运行界面如图 12.8 所示。

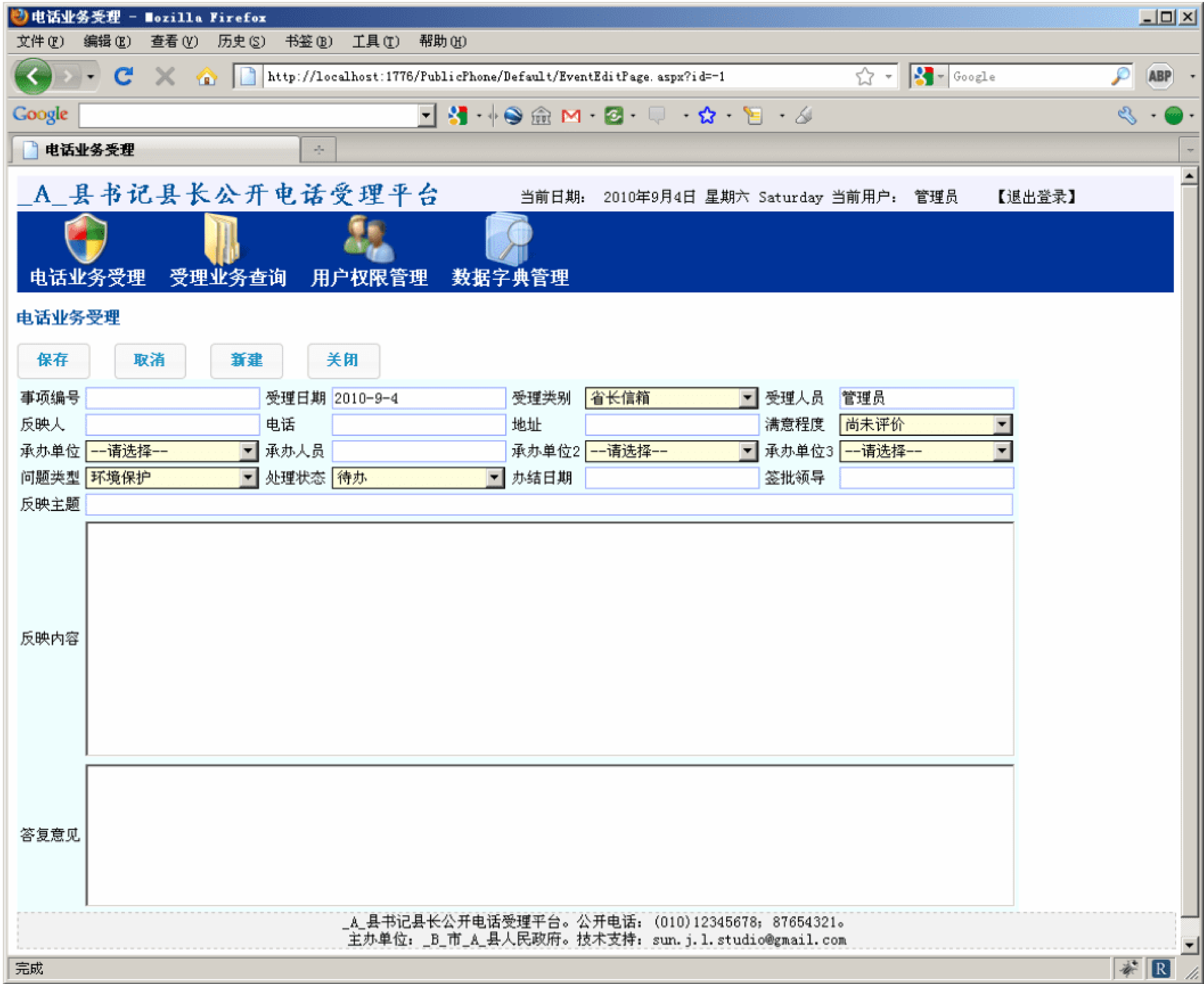


图 12.8　电话业务受理页面

12.4　电话业务综合查询

在县长公开电话受理系统中，用户可以对所有群众电话反映的事件进行各种查询，查询条件包括受理日期、反映人姓名、承办单位、承办类型等，用户可以根据以上条件中的一个或者任意多个组合进行查询，并对查询出的事件进行一些操作，如编辑、打印等。本节将介绍电话业务综合查询功能的设计和实现。

12.4.1　通用组合条件查询

在各种数据库应用程序中，都会遇到组合条件查询，即根据某些条件的任意组合对某一数据源进行查询。在以前的编程语言中，很难编写一个强类型的通用组合查询方法。

在.NET 4.0 中，利用泛型、Lambda 表达式、LINQ、实体框架等技术，就可以设计出一个适用于各个数据实体的、接受任意查询条件且支持分页功能的通用组合条件查询。下面将介绍这种查询的设计思路和实现代码，下面的讨论以公开电话受理系统的事件查询为例。

为了设计出通用的组合条件查询方法，首先要对各种查询及其实现进行分析，然后寻找各种查询的共同点和不同点，对共同点进行抽象，不同点进行封装，进而实现通用设计。下面的讨论就是按照这个思路进行的。

1．简单查询

最简单的查询是查询所有数据，用代码描述如下：

```
public List<EventInfo> getAll()
{
    var context = new UserRightContext();      //创建 ObjectContext 对象
    context.EventInfo.MergeOption = MergeOption.NoTracking;
                                               //不需要跟踪对象状态
    var query = from e in context.EventInfo
                select e;                      //定义查询
    var list = query.ToList();                 //执行查询，得到结果
    context.Dispose();                         //释放资源
    return list;                               //返回结果
}
```

2．单条件查询

单条件查询可以根据某一条件进行查询，如根据受理人员、受理日期、事件编号等进行查询。实现此功能的代码如下：

```
/// <summary>
/// 根据受理人员查询事件详情
/// </summary>
/// <param name="handler">受理人员</param>
public List<EventInfo> getByHandler(string handler)
{
    var context = new UserRightContext();      //创建 ObjectContext 对象
    context.EventInfo.MergeOption = MergeOption.NoTracking;
                                               //不需要跟踪对象状态
     //创建 LINQ 查询
    var query = from e in context.EventInfo
                where e.HandleUser==handler
                select e;
    var list = query.ToList();                 //执行查询，得到结果
    context.Dispose();                         //释放资源
    return list;                               //返回结果
}
/// <summary>
/// 根据受理时间查询事件详情
/// </summary>
///<param name="begin">查询开始时间</param>
///<param name="end">查询结束时间</param>
public List<EventInfo> getByHandler(DateTime begin, DateTime end)
{
    var context = new UserRightContext();      //创建 ObjectContext 对象
```

```
    context.EventInfo.MergeOption = MergeOption.NoTracking;
                                                        //不需要跟踪对象状态
    //定义 LINQ 查询
    var query = from e in context.EventInfo
                where e.HandleDate>=begin && e.HandleDate<=end
                select e;
    var list = query.ToList();                          //执行查询，得到结果
    context.Dispose();                                  //释放资源
    return list;                                        //返回结果
}
```

上述代码仅实现了根据受理人和受理日期进行查询的方法。在实际项目中，要查询的条件远不止这两种，用这种思路实现查询，每个查询都需要写一个方法，则类中会包含很多类似的方法。

3．组合条件查询

组合条件查询是指按照两个以上的不同条件组合进行查询，例如查询受理人张三在一月份受理的所有事件，或者查询受理人张三受理的所有环境保护类事件情况等，示例代码如下：

```
public List<EventInfo> getByHandlerAndDate(string handler,DateTime begin,
DateTime end)
{
    var context=new UserRightContext();
    context.EventInfo.MergeOption = MergeOption.NoTracking;
    //定义 LINQ 查询
    var query=from e in context.EventInfo
              where e.HandleDate>=begin
              && e.HandleDate<=end
              && e.HandleUser==handler
              select e;
    var list = query.ToList();                              //执行查询，得到结果
    context.Dispose();                                      //释放资源
    return list;
}
```

如果查询条件共有 N 种，那么各种条件组合可达到 N 的阶乘，按照这种思路写代码，要想实现所有条件组合几乎是不可能的。

4．分页查询

在真实项目中，由于数据量巨大，绝大多数查询都是分页查询，每次仅需返回指定页码的数据，示例代码如下：

```
public List<EventInfo> pageQuery()
{
    int pageIndex = 3;                                  //页码
    int pageSize = 10;                                  //页面大小
    var context = new UserRightContext();
    context.EventInfo.MergeOption = MergeOption.NoTracking;
    var query = from e in context.EventInfo
                where e.HandleDate >= DateTime.Parse("2010-1-1")
                && e.HandleDate <= DateTime.Parse("2010-1-31 23:59:59")
                && e.HandleUser == "admin"
```

```
                orderby e.EventID                        //排序
                select e;
    var list = query.Skip((pageIndex-1)*pageSize).Take(pageSize).
                                                         ToList();//执行分页查询
    context.Dispose();                               //释放资源
    return list;
}
```

综合分析上述各种查询，可以得出以下结论：

（1）各种查询的共同点是都需要创建 ObjectContext，都不用跟踪对象状态，都返回一个实体列表，都需要分页（不分页可视为页面大小极大的特殊情况）。

（2）各种查询的不同点是返回的实体类型不同，用到的 ObjectQuery 查询对象不同，查询条件不固定，排序方法不一样。

如果可以编写一个查询方法，此方法可以指定查询所使用的 ObjectQuery 类型、返回的实体类型、任意数量的查询条件、排序关键字和升降序，则可以实现通用组合条件查询。其中类型参数可用泛型实现，查询条件可用 Lambda 表达式实现，排序表达式也用 Lambda 表达式实现。按照这个思路，编写通用组合条件查询方法如下：

```
/// <summary>
/// 通用组合条件查询
/// </summary>
/// <typeparam name="T">实体对象类型</typeparam>
/// <typeparam name="TSortKey">排序关键字类型</typeparam>
/// <param name="query">ObjectQuery 对象</param>
/// <param name="sort">排序表达式</param>
/// <param name="sortDesc">是否降序排序</param>
/// <param name="page">分页参数</param>
/// <param name="dispose">查询结束是否释放 ObjectContext 资源</param>
/// <param name="predicate">查询条件</param>
/// <returns>查询结果列表</returns>
internal static List<T> selectMany<T,TSortKey>(ObjectQuery<T> query,
Expression<Func<T,TSortKey>> sort, bool sortDesc,
    PageDataArgument page, bool dispose,
    params Expression<Func<T,bool>>[] predicate)    //0 到多个查询条件
    where T:EntityObject
{
    query.MergeOption = MergeOption.NoTracking;     //不需要跟踪对象状态
    IQueryable<T> q = query;                        //查询对象
    for (int i = 0; i < predicate.Length; i++)      //逐个添加查询条件
    {
        q = q.Where(predicate[i]);
    }
    if (sortDesc)                                   //添加排序条件
        q = q.OrderByDescending(sort);
    else
        q = q.OrderBy(sort);
    if (page.refreshCount)                          //刷新记录总数
        page.count = q.Count();
    var list = q.Skip(page.pageIndex * page.pageSize).Take(page.pageSize).
    ToList();
    detachEntity(query.Context, list);              //分离实体对象
    if (dispose)
        query.Context.Dispose();
    return list;
```

```
}
/// <summary>
/// 将列表中的实体对象从 ObjectContext 中分离
/// </summary>
/// <typeparam name="T">实体对象类型</typeparam>
/// <param name="context">ObjectContext</param>
/// <param name="list">对象列表</param>
internal static void detachEntity<T>
    (ObjectContext context, List<T> list)
    where T : EntityObject
{
    list.ForEach(item =>detachAndRemoveEntityKey(context, item));
}
//从对象上下文分享对象并删除实体键
private  static  void  detachAndRemoveEntityKey(ObjectContext  context,
EntityObject entity)
{
    if(entity.EntityState!=EntityState.Detached)
        context.Detach(entity);
    entity.EntityKey = null;
}
```

🔔**提示**：selectMany()方法的最后一个参数前有一个 params 关键字，表示这是数目可变的参数，即可以接受 0 到多个查询条件。

上述代码中 selectMany()方法参数太多，调用不方便且不易阅读。可以把这些参数封装到一个类 ObjectQueryArgument 中，为各个参数设置默认值，这样就能够使 selectMany()方法的使用更加简洁。ObjectQueryArgument 类代码如下：

```
/// <summary>
/// 通用组合查询参数，在 EntityUtility.selectMany()方法中使用
/// </summary>
/// <typeparam name="T">实体对象类型</typeparam>
/// <typeparam name="TSortKey">排序关键字类型</typeparam>
internal class ObjectQueryArgument<T,TSortKey>
    where T:IEntityWithKey
{
    #region 构造函数
    internal ObjectQueryArgument()
    {
        disposeAfterQuery = true;
        descending = false;
    }
    internal ObjectQueryArgument(ObjectQuery<T> theQuery, Expression
    <Func<T,TSortKey>> sortExp,PageDataArgument pageArg)
    {
        descending=false;
        query=theQuery;
        sort=sortExp;
        page=pageArg;
    }
    #endregion
    #region 属性
    internal ObjectQuery<T> query { get; set; }
    internal Expression<Func<T, TSortKey>> sort{get;set;}
    internal bool descending { get; set; }
    internal PageDataArgument page { get; set; }
```

```
    //执行完查询后是否释放 ObjectContext
    internal bool disposeAfterQuery { get; set; }
    #endregion
}
```

使用 ObjectQueryArgument 类作为参数的 selectMany()方法如下：

```
internal static List<T> selectMany<T, TSortKey>
    (ObjectQueryArgument<T, TSortKey> arg,            //查询参数
    params Expression<Func<T, bool>>[] predicate)     //查询条件
    where T:EntityObject                              //泛型约束
{
    return selectMany<T, TSortKey>
        (arg.query, arg.sort, arg.descending, arg.page, arg.disposeAfter-
        Query, predicate);
}
```

12.4.2　电话业务综合查询数据层和业务层

通过与用户交流，电话业务查询主要根据以下几个条件：受理日期、事件编号、反映人姓名、责任单位、事件类型、事件来源。基于 12.4.1 节所设计的通用组合条件查询方法，能够实现电话业务综合查询功能，具体实现步骤如下：

（1）在数据访问层项目 SJL.Dal 的 EventInfoDal 类中，添加一个 search()方法，实现综合查询功能，代码如下：

```
/// <summary>
/// 综合查询
/// </summary>
/// <param name="page">分页参数</param>
/// <param name="predicates">查询条件</param>
/// <returns>查询结果</returns>
public static List<EventInfo> search(PageDataArgument page,
    params Expression<Func<EventInfo,bool>>[] predicates)
{
    var arg = new ObjectQueryArgument<EventInfo, DateTime?>();
    arg.query = getQuery();
    arg.sort = e => e.HandleDate;                     //按照受理日期排序
    arg.descending = true;                            //降序排序
    arg.disposeAfterQuery = false;                    //不立即释放资源
    var list= EntityUtility.selectMany(arg,predicates);//执行查询
    list.ForEach(item => loadDetail(item));           //加载外键数据
    arg.query.Context.Dispose();                      //释放资源
    return list;
}
```

（2）在 EventInfoDal 类中添加一个 loadReference()方法，用于加载事件信息的外键数据，如事件来源、责任部门等：

```
/// <summary>
/// 加载事件外键数据
/// </summary>
/// <param name="info">要加载的事件</param>
private static void loadReference(EventInfo info)
{
```

```
    if (info.EventSource.HasValue)
        info.eventSourceName = info.EventSource1.Name;
    if (info.TypeID.HasValue)
        info.typeName=info.EventType.Name;
    if (info.StatusID.HasValue)
        info.statusName = info.EventStatus.Name;
    if (!string.IsNullOrEmpty(info.ResponsibleDepartment))
        info.responsibleDepartmentName = info.Department.DepartmentName;
    info.Department1Reference.Load();
    info.Department2Reference.Load();
}
```

（3）目前为止，实现的综合查询 search()方法需要传递多个 Lamba 表达式，方法调用不够简洁，可以添加一个重载的 search()方法用更清晰、直观的方式实现同样的查询功能。为此，需要添加一个类 EventSearchCondition 封装事件综合查询条件，然后将此类作为参数传递给 search()方法。代码如下：

```
/// <summary>
/// 事件查询条件
/// </summary>
public class EventSearchCondition
{
    public string id { get; set; }                          //事件编号
    public int source { get; set; }                         //事件来源
    public int type { get; set; }                           //事件类型
    public string department { get; set; }                  //责任部门
    public string person { get; set; }                      //反映人姓名
    public DateRange date { get; set; }                     //日期范围
}
/// <summary>
/// 综合查询
/// </summary>
/// <param name="condition">查询条件</param>
/// <param name="page">分页参数</param>
/// <returns>查询结果</returns>
public static List<EventInfo> search(EventSearchCondition condition,
PageDataArgument page)
{
    List<Expression<Func<EventInfo, bool>>> predicates
        = new List<Expression<Func<EventInfo, bool>>>();    //查询条件列表
    //如果时间范围不为空，则添加这个查询条件
    if (condition.date != null)
        predicates.Add(
            e => e.HandleDate >= condition.date.from
                && e.HandleDate < condition.date.to);
    //如果责任部门不为空，则添加此条件
    if (!string.IsNullOrEmpty(condition.department))
        predicates.Add(e => e.ResponsibleDepartment == condition.
        department);
    //如果反映人不为空，则添加此条件
    if (!string.IsNullOrEmpty(condition.person))
        predicates.Add(e => e.PersonName == condition.person);
    //如果事件来源不为空，则添加此条件
    if(condition.source>0)
        predicates.Add(e => e.EventSource == condition.source);
    //如果事件类型不为空，则添加此条件
    if(condition.type>0)
```

```
        predicates.Add(e => e.TypeID == condition.type);
    if (predicates.Count == 0)
        return search(page);
    return search(page, predicates.ToArray());
}
```

（4）在业务逻辑层项目 SJL.Bll 的 EventInfoBll 类中，添加一个 search()方法，代码如下：

```
/// <summary>
/// 综合查询
/// </summary>
/// <param name="condition">查询条件</param>
/// <param name="page">分页参数</param>
/// <returns>查询结果</returns>
public  static  List<EventInfo>  search(EventSearchCondition  condition,
PageDataArgument page)
{
    //如果事件编号不为空，则只根据事件编号查询（忽略其他条件）
    if (!string.IsNullOrEmpty(condition.id))
    {
        List<EventInfo> list=new List<EventInfo>();
        var item=getByID(condition.id);
        if(item!=null)
            list.Add(item);
        return list;
    }
    return DB.search(condition, page);              //执行综合查询
}
```

12.4.3　事件列表控件

在公开电话受理系统中有多个需要显示事件信息的列表，为了实现界面和代码复用，可以用一个用户控件 EventListControl.ascx 封装事件列表功能，然后在需要显示事件列表的页面上直接使用此用户控件。具体实现步骤如下：

（1）在表现层项目 SJL.Web 的 UserControls 文件夹中添加一个 EventListControl.ascx 用户控件。

（2）在用户控件 EventListControl.ascx 中放置一个 GridView，以列表形式显示事件主要信息，并在 GridView 最后几列添加编辑和打印按钮，如图 12.9 所示。



图 12.9　事件列表用户控件

用户控件 EventListControl.ascx 代码如下：

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="EventList-
```

```
Control.ascx.cs" Inherits="SJL.Web.UserControls.EventListControl" %>
 <script type="text/javascript">
     //打开编辑对话框
     function openEditDialog(id) {
         openMyDialog("../Default/EventEditSmallPage.aspx", "?id=" + id);
     }
     //打开打印对话框
     function openPrintDialog(id) {
         window.showModalDialog("../Default/EventPrintPage.aspx"+"?id=" +
         id);
     }
     //打开一个浏览器模式对话框
     function openMyDialog(url, querystring) {
         window.showModalDialog(url+querystring, null,
            "dialogWidth:960; dialogHeight:500; center:yes; menubar:yes;");
     }
 </script>

<asp:GridView ID="grid1" CssClass="gridview" runat="server"
    DataKeyNames="EventID" AutoGenerateColumns="false" >
<Columns>
<asp:BoundField DataField="EventID" HeaderText="受理编号" ItemStyle-Width=
"80" />
<asp:BoundField DataField="HandleDate" DataFormatString="{0:d}" Header-
Text="受理日期" ItemStyle-Width="100" />
<asp:BoundField DataField="PersonName" HeaderText="反映人" ItemStyle-
Width="70"/>
<asp:BoundField DataField="EventSourceName" HeaderText="受理来源" ItemSty-
le-Width="100" />
<asp:BoundField DataField="TypeName" HeaderText="受理类型"  ItemStyle-
Width="100"/>
<asp:BoundField DataField="ResponsibleDepartmentName" HeaderText="承办单位
" ItemStyle-Width="120" />
<asp:BoundField  DataField="StatusName" HeaderText="处理情况" ItemStyle-
Width="60" />
<asp:BoundField DataField="EvaluationName" HeaderText="满意度" ItemStyle-
Width="80" />
<asp:TemplateField HeaderText="详情">
<ItemTemplate>
<a href="#" onclick="openEditDialog('<%#Eval("EventID")%>');" > <%--编辑
链接--%>
<img src="../images/edit.png" alt="" class="noborder"/> </a>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="打印单据">                <%--打印链接--%>
<ItemTemplate>
<a href="../Default/BanLiPrintPage.aspx?id=<%#Eval("EventID")%>" target=
"eventPrintWindow" >办理</a>
<a href="../Default/CBPrintPage.aspx?id=<%#Eval("EventID")%>" target=
"eventPrintWindow" >承办</a>
<a href="../Default/FKPrintPage.aspx?id=<%#Eval("EventID")%>" target=
"eventPrintWindow" >反馈</a>
</ItemTemplate>
</asp:TemplateField>
</Columns>
 </asp:GridView>
```

（3）在用户控件 EventListControl.ascx.cs 中编写一个方法，将数据绑定到 GridView：

```
public void bindList(List<EventInfo> list)
{
    grid1.DataSource = list;
    grid1.DataBind();
}
```

12.4.4　综合查询页面

在综合查询页面中，用户可以输入各种查询条件并执行查询，还可以对查询结果进行编辑、打印等操作。实现综合查询页面的具体操作步骤如下：

（1）在表现层项目 SJL.Web 中，从母版页 SjlMaster.master 创建一个新的页面 EventListPage.aspx，在页面上放置多个文本框控件以允许用户输入查询条件，使用 jQuery UI 中的日历控件输入日期，放置一个 EventListControl 用户控件以显示查询结果，如下代码所示：

```
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <script type="text/javascript">
        $(function() {
            initControls();
        });    //$(function)
        function initControls() {
            //为 GridView 添加光棒效果
            $('table.gridview').find("tr").hover(
                function() { $(this).addClass('hoverRow'); },
                function() { $(this).removeClass('hoverRow'); }
            ); //$('table').tr.hover
            $SjlUtility.addButtonClass();         //为按钮添加样式
            $SjlUtility.jQueryDatePickerChinese();
                                                   //初始化 jQuery UI 日历控件
            $('#<%=date1.ClientID%>').datepicker();
                                                   //为 date1 控件启用 jQuery UI
            $('#<%=date2.ClientID%>').datepicker();
                                                   //为 date2 控件启用 jQuery UI
            $('#printbutton').click( printGrid );
                                                   //为打印按钮添加事件处理程序
        }
        function printGrid() {
            window.open("EventListPrint.aspx");
        }
</script>
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="content" runat="server">
    <h3>受理业务列表</h3>
    事件编号：<asp:TextBox ID="eventID" runat="server"></asp:TextBox>
    受理日期：<asp:TextBox ID="date1" runat="server" ></asp:TextBox>
    至<asp:TextBox ID="date2" runat="server"></asp:TextBox>
    事件来源：<asp:DropDownList ID="sourceList" runat="server"></asp:
    DropDownList><br />
    承办类型：<asp:DropDownList ID="typeList" runat="server"></asp: Drop-
    DownList>
    承办单位：<asp:DropDownList ID="departmentList" runat="server"
    DataValueField="departmentID" DataTextField="DepartmentName"></asp:
    DropDownList>
```

```
    反映人：<asp:TextBox ID="person" runat="server"></asp:TextBox>  
    <asp:Button ID="clear" runat="server" Text="清空条件" onclick="clear_
    Click" />
     <asp:Button ID="Button1" runat="server" Text="执行查询" onclick=
    "Button1_Click" />
     <input type="button" value="打印表格" id="printbutton" />
    <uc1:EventListControl ID="list1" runat="server" />
    <webdiyer:AspNetPager ID="pager1" runat="server"
        onpagechanged="pager1_PageChanged" PageSize="20">
    </webdiyer:AspNetPager>
</asp:Content>
```

（2）在 EventListPage.aspx 页面的 Page_Load 事件中将数据绑定到查询条件中的下拉列表中，包括责任单位列表、事件来源列表和事件类型列表：

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        initList();
    }
}
//绑定过滤条件中的下拉列表
private void initList()
{
    string all="--全部--";
    //绑定事件来源列表
    WebUtility.bindListWithDictionary<EventSourceBll,
EventSource>(sourceList);
    typeList.Items.Insert(0,new ListItem(all,"-1"));
    //绑定事件类型列表
    WebUtility.bindListWithDictionary<EventTypeBll, EventType>(typeList);
    sourceList.Items.Insert(0,new ListItem(all,"-1"));
    //绑定责任单位列表
    var list = DepartmentBLL.getAll(PageDataArgument.allData);
    departmentList.DataSource = list;
    departmentList.DataBind();
    departmentList.Items.Insert(0,new ListItem(all, ""));
}
```

（3）在“执行查询”按钮的 Click 事件中，根据查询条件执行查询并显示结果：

```
protected void Button1_Click(object sender, EventArgs e)
{
    pager1.CurrentPageIndex = 1;
    bindData();
}
private void bindData()
{
    var list = executeQuery();
    list1.bindList(list);
}
/// <summary>
/// 根据用户输入的查询条件执行查询
/// </summary>
/// <returns>查询结果</returns>
private List<SJL.Entity.EventInfo> executeQuery()
{
    PageDataArgument page= new PageDataArgument(pager1.CurrentPage
```

```
    Index - 1,
        pager1.PageSize, true);
    //以下语句设置查询条件
    EventSearchCondition condition = new EventSearchCondition();
    condition.id = eventID.Text;
    condition.date = DateRange.between2Date(date1.Text, date2.Text);
    condition.department = departmentList.SelectedValue;
    condition.person = person.Text;
    condition.source = int.Parse(sourceList.SelectedValue);
    condition.type = int.Parse(typeList.SelectedValue);
    var list= EventInfoBLL.search(condition, page);        //执行查询
    Session["EventListPrintData"] = list;
    pager1.RecordCount = page.count;
    return list;
}
```

（4）在分页控件的 PageChanged 事件中重新绑定新页面数据：

```
protected void pager1_PageChanged(object sender, EventArgs e)
{
    bindData();
}
```

（5）在浏览器中查看 EventListPage.aspx 页面，运行界面如图 12.10 所示。



图 12.10　电话业务综合查询页面

12.5　报 表 打 印

在县长公开电话受理系统中，需要产生各种单据和表格，这些单据和表格需要打印出来，让相关人员浏览、签字并存档。本节将介绍如何生成和打印这些表格。

12.5.1　报表母版页

通过对县长公开电话受理系统中所有表格进行分析，发现各种单据和表格拥有一些共

同点：都需要打印到 A4 纸张上，都有一个表头，表格字体大小要求可以调整。为了实现复用，可以设计一个专用于打印报表的母版页，把这些通用功能放在母版页中。报表母版页的具体实现步骤如下：

（1）在公开电话受理系统的表现层项目 SJL.Web 中添加一个新的母版页，命名为 ReportMaster.master。

（2）在母版页上放置一个 div，用 CSS 将其大小、边距等设置为与 A4 纸张相适应：

```
<style type="text/css">
.a4{ margin-left:10px; margin-top:10px; width:640px; font-size:16px; }
</style>
<div class="a4">
</div>
```

（3）在页面上放置一个 Label 以显示报表标题，放置一个 ContentPlaceHolder 以在内容页中显示具体报表内容。

```
<div class="a4">
    <div id="thetitle" class="reporttitle">
    <asp:Label ID="reportTitle" runat="server" Text="书记、县长公开电话受理事
    项承办单"> </asp:Label>
    </div>
    <asp:ContentPlaceHolder ID="content" runat="server">
    </asp:ContentPlaceHolder>
    </div>
</div>
```

（4）在母版页顶部放置用于打印、设置表头和调整字体大小的控件：

```
<div class="noprint">
<input type="button" value="打印" onclick="window.print();" />
<input type="button" value="编辑" id="editButton" />
表头：<input type="text" id="titleText"  style="width:300px;" value="书记、
县长公开电话受理事项 XX 单" />
<input type="button" value="设置表头" id="titleButton" />
 字体大小：<select id="fontSizeOption" style="width:auto;">
<option value="12">1</option>
<option value="14">2</option>
<option value="16" selected>3</option>
<option value="18">4</option>
<option value="20">5</option>
</select>
<br />
<span class="obvious">提示：如需设置打印比例、页边距等，请从浏览器相应菜单中进行操
作。</span>
</div>
```

报表母版页设计界面如图 12.11 所示。

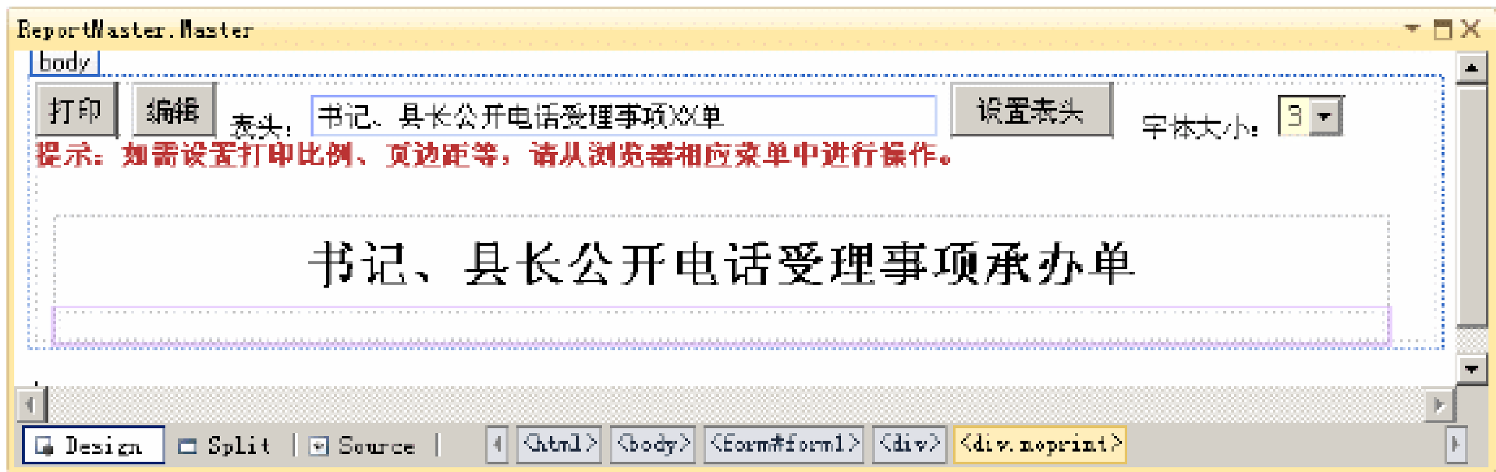


图 12.11　报表母版页设计界面

（5）母版页顶部的按钮、文本框等控件仅在浏览页面时可见，在打印时不可见，可以通过 CSS 实现这个效果，CSS 代码如下：

```
<style type="text/css" media="print">          //此样式仅对打印机有效
.noprint{display:none;}                          //打印时不显示
</style>
```

（6）报表母版页中的打印、设置表头、修改字体等功能都可以在浏览器端实现而无须发回服务器进行处理，所以为相应控件添加 JavaScript 脚本以实现这些功能：

```
<script type="text/javascript">
   $(function() {
      $('#editButton').click(openEditDialog);          //打开编辑对话框
      $('#titleButton').click(setTitle);               //设置报表标题
      $('#fontSizeOption').change(changeFont);         //改变字体大小
   });
   //设置报表标题
   function setTitle() {
      $('div#thetitle > span').html($('#titleText').val());
   };
   //打开编辑对话框
   function openEditDialog() {
      var id = $('#hiddenid').val();
      window.showModalDialog("../Default/EventEditSmallPage.aspx?id=" +
      id,
      null, "dialogWidth:900;dialogHeight:500;center:yes;menubar:
      yes;");
      window.location = window.location;
   };
   //改变字体大小
   function changeFont() {
      var size = $(this).val();
      $('div.a4').css("font-size", size+'px');
   };
</script>
```

12.5.2 打印承办单

县长公开电话受理系统中的承办单反映了群众所反映事件的承办情况，此单据需将事件的基本信息打印出来，交由相关领导签字。生成和打印承办单的具体步骤如下：

（1）在表现层项目 SJL.Web 中，基于报表母版页 ReportMaster.master 添加一个新页面 CBPrintPage.aspx。

（2）在页面上放置一个 table，并按照用户提供的样表设计表格的内容和格式。页面代码如下：

```
<asp:Content ID="content2" ContentPlaceHolderID="content" runat="server">
<input type="hidden" id="hiddenid" value="<%=eventId.Text%>" /> <%--事件
ID--%>
   <table  style="border:none;" width="100%" rules="none" >
   <tr>
   <td>事项编号：
   <asp:Label runat="server" ID="eventId" Text="" ReadOnly="true"></asp:
   Label></td>
```

```
        <td>办理时限：10 个工作日</td>
        <td> 受理日期:
        <asp:Label ID="theDate" runat="server" Text="" ReadOnly="true">
        </asp: Label></td>
    </tr>
    </table>
    <br />
     <!--表格总高度：750px,不含表头、表尾和第 1 行-->
     <table rules="all"  border="1"  width="100%" cellpadding="4" style=
     "text-align:center;">
    <tr><td class="colheader" style="height:60px;">来电<br />
    人员</td><td>
    <asp:Label ID="personName" runat="server"></asp:Label>
    </td>
    <td class="colheader">联系<br />
    电话</td><td>
    <asp:Label ID="personPhone" runat="server"></asp:Label>
    </td><td class="colheader">
    联系<br />
        地址</td><td>
    <asp:Label ID="personAddress" runat="server"></asp:Label>
    </td> </tr>
<tr> <td>内<br />容<br />提<br />要<br /></td>
<td colspan="5" style="height:200px; vertical-align:top; text-align:
left;"> <br />
        <asp:Label ID="eventContent" runat="server"  />
    </td>    </tr>
    <tr>   <td>
    <!--使用 pre 标记实现预排版功能，为了对齐格式，下面的 8 个字需要显示在 4 行-->
    <pre style="font-size:16px;">
    县室
    政领
    府导
    办意
    公见
    </pre>
    </td><td colspan="5" style="height:180px; vertical-align:top; text-
    align: left;"> </td>
    </tr>
    <tr> <td >领<br /> 导<br /> 批<br /> 示 </td>
    <td colspan="5" style="height:250px;">
    </td> </tr>
    <tr><td>办<br />理<br />结<br />果</td><td colspan="5" style="height:
    120px;">
    </td> </tr>
</table>
<div style="margin-right:20px; text-align:right;">
受理人：<asp:Label runat="server" ID="personList2"  Width="120" />
初审人：<asp:Label runat="server" ID="label01" Width="120" />
审核人：<asp:Label runat="server" ID="label02" Width="120" />
</div>
</asp:Content>
```

（3）在页面的 Load 事件中，设置报表标题和报表内容。由于报表标题控件在母版页中，所以要使用 Master.FindControl 找到标题所对应的 Label 控件：

```
protected void Page_Load(object sender, EventArgs e)
```

```
{
    if (!IsPostBack)
    {
        Label l=this.Master.FindControl("reportTitle") as Label;
                                                  //找到母版页中的标题控件
        l.Text = "书记、县长公开电话受理事项承办单";      //设置标题文字
        string s = Request.QueryString["id"];        //得到要显示的事件 ID
        setByID(s);
    }
}
/// <summary>
/// 根据事件 ID 设置报表内容
/// </summary>
/// <param name="id">事件 ID</param>
private void setByID(string id)
{
    if (string.IsNullOrEmpty(id))
        return;
    EventInfo info = EventInfoBLL.getByID(id);      //根据 ID 得到事件详情
    eventId.Text = info.EventID;
    if (info.HandleDate.HasValue)
        theDate.Text = info.HandleDate.Value.ToShortDateString();
    //根据事件详情设置各个控件的值
    personName.Text = info.PersonName;
    eventContent.Text = info.EventContent.Replace("\n", "<br/>");
    personList2.Text = info.HandleUser;
    personAddress.Text = info.PersonAddress;
    personPhone.Text = info.PersonPhone;
}
```

（4）运行承办单页面 CBPrintPage.aspx，运行页面如图 12.12 所示。

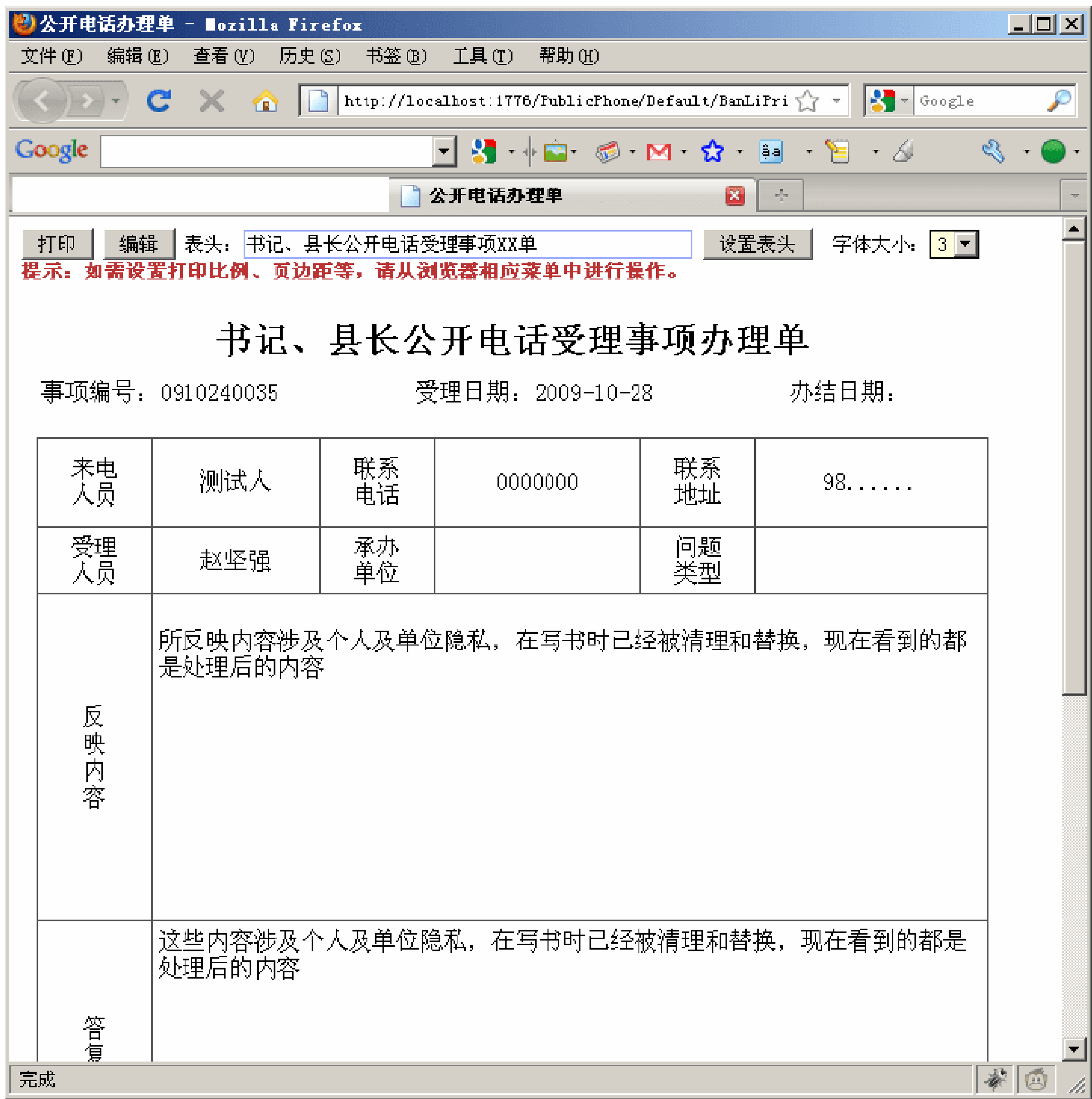


图 12.12　承办单页面

在图 12.12 所示的承办单页面中单击“打印”按钮，弹出如图 12.13 所示的“打印”对话框。

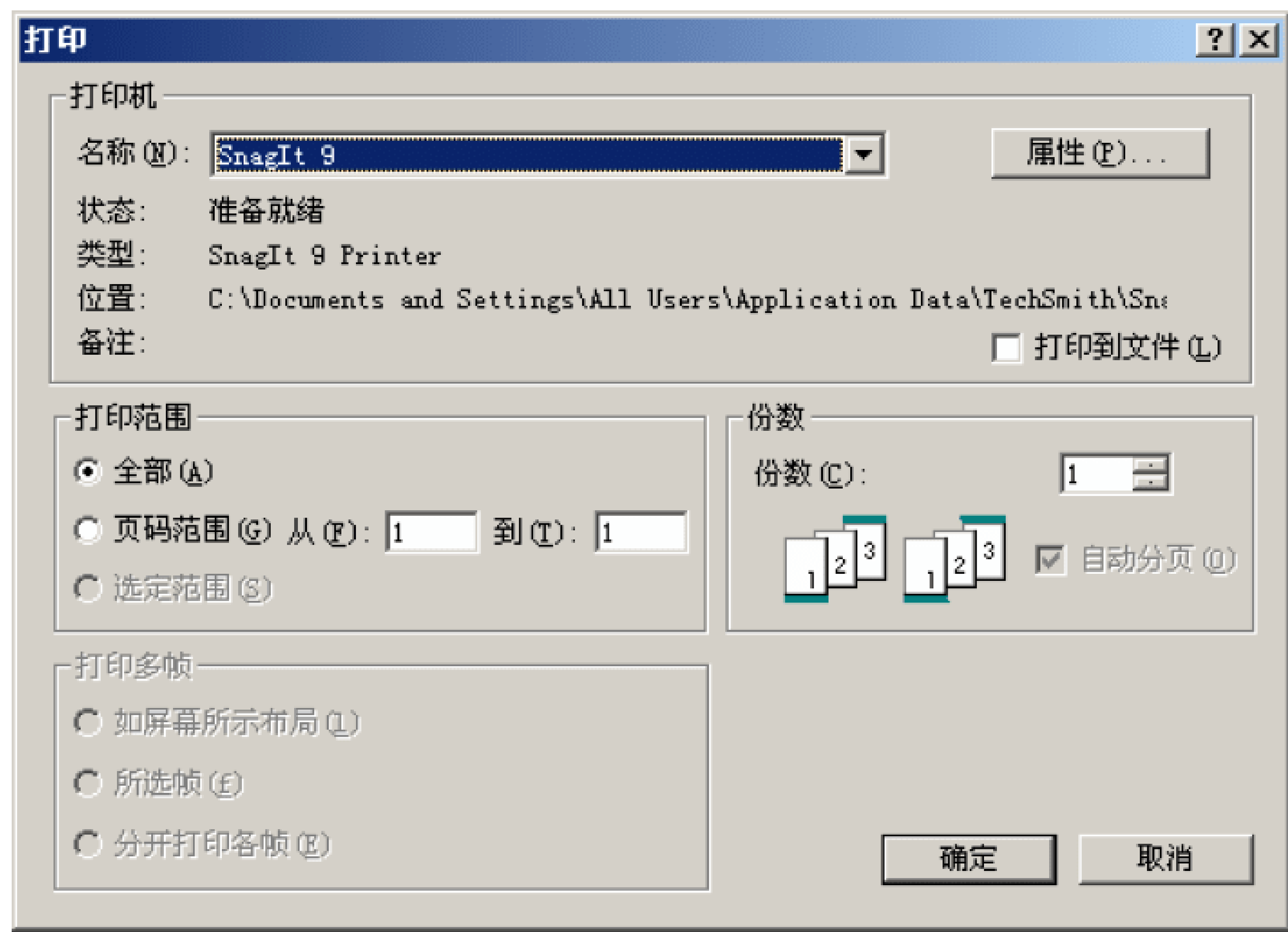


图 12.13　浏览器“打印”对话框

12.6　小　　结

本章介绍了 A 县县长公开电话受理系统的设计与实现，首先介绍了用户需求和系统整体设计思路，然后重点讲解了业务受理、综合查询和报表打印等功能的设计和实现。本系统规模较小，是学习 ASP.NET 的一个好案例。本系统已经在滨州市 A 县得到成功应用，对于类似系统的开发具有一定借鉴意义。

第 13 章　社保卡结算系统

本章所介绍的案例是作者为某市人力资源和社会保障局开发的一个软件。出于安全和隐私方面的原因，书中不便给出该市的真实名称，下文均以 A 市代替。本系统以社保卡开户银行的数据为基础，结合 A 市人力资源和社会保障局社保管理系统的数据，对社保卡消费数据进行各种处理，可以实现 A 市社保卡的资金结算和数据统计功能。

13.1　整体设计思路

本节将介绍 A 市社保卡结算系统的开发背景和业务流程，分析用户需求，提出此系统整体设计思路，实现数据库结构设计。

13.1.1　项目简介

A 市在全市范围内推行了社会保障卡发行工作，参保人员持社会保障卡即可在全市统一定点的医疗机构网络内无障碍就医、就诊和结算，实现全市医疗服务“一卡通”。社会保障卡是按照劳动和社会保障统一规划，由各地劳动社会保障部门面向城镇从业人员和离退休人员发行的用于劳动和社会保障业务的集成电路卡。社保卡的发放范围为全市城镇职工医疗、养老、失业、工伤和生育保险的参保人员，和符合参加城镇居民基本医疗保险条件的居民。社会保障卡可用于查询个人基本信息和养老、医疗保险个人账户，办理住院和结算手续，进行医疗保险个人账户消费。

社保卡由参保人员使用，在各个医疗机构进行消费，是一种银行卡，所有社保卡的消费信息都在银行数据库中有记录。银行每隔一定时间（通常是一个月）向 A 市人力资源和社会保障局提供一个社保卡消费明细（dbf 格式），A 市人力资源和社会保障局根据银行提供的数据与社保管理系统数据库中的数据进行对比，并在核对之后将资金结算给相应的医疗机构。

A 市人力资源和社会保障局现在有一个社保管理系统用于管理社保相关数据，如参保人员信息、医疗机构信息、社保机构信息等，社保管理系统的服务器位于 A 市人力资源和社会保障局，服务器操作系统为 Windows，数据库管理系统为 Oracle。社保卡结算系统需要运行在这台服务器上，并使用服务器上原有数据，结合银行提供的社保卡消费明细，生成结算数据及各种统计报表。

银行的程序与社保程序是完全独立的两个系统，完成不同的数据库结构，同一个医疗机构在银行的数据库中，与社保程序数据库中具有完全不同的编码，在设计开发社保卡结算系统时要注意数据接口和编码对应。综上所述，社保卡结算系统主要需求如下。

（1）系统运行在 A 市人力资源和社会保障局现有服务器上，使用现有 Oracle 数据库。在选用开发工具时，需要考虑与操作系统和数据库的兼容性。

（2）本系统的用户有两类：A 市人力资源和社会保障局人员，医疗机构人员。两类用户具有不同的权限。

（3）实现银行数据库中医疗机构编码与社保管理系统数据库中医疗机构编码之间的对应关系。

（4）将银行提供的 dbf 格式的社保卡消费数据导入到数据库中。

（5）对从银行导入的数据进行审核。

（6）对各个医疗机构进行结算，并保存结算相关数据。结算可以分批进行，例如，应该付给某医疗机构 10 万元，可以分几次将款结清。

（7）按照社保机构和日期查询社保卡消费情况，生成各种报表。

13.1.2　数据库结构

社保卡结算系统涉及三种类型的数据表，首先是社保管理系统中原来的数据表，第二是银行提供的社保卡消费明细 dbf 数据，第三是为了社保卡结算系统自行设计和添加的表。其中前两者表结构已经确定，程序只需要使用即可。银行提供的社保卡消费明细数据表（dbf 格式）包括以下字段。

- ID：记录编号，char 类型，主键。
- WordDate：交易日期，date 类型。
- Corno：商户号，char 类型，即医院或者药店在银行的客户编号。
- Name：商户名称，char 类型，即医院或者药店名称。
- Accno：银行卡号，char 类型，即社保卡号。
- Amt：消费金额，Numeric 类型。
- Fee：银联交易手续费，Numeric 类型，按照消费金额的一定比例收取手续费。
- Realamt：实际结算金额，Numeric 类型，社会保障局将按照这个金额与医疗机构进行结算。
- Type：消费类型，char 类型，为 PCA 正常、PPT 退货、PCC 消费冲正和 PPC 退货冲正这 4 个值之一。
- Term：终端机号，char 类型，银联刷卡的终端机号。
- Time：消费时间，date 类型，此笔消费所发生的具体时间。
- Cserno：银联交易流水号，char 类型。

在 A 市人力资源和社会保障局的 Oracle 数据库服务器上，与社保卡消费相关的有大约 10 个表，为了便于开发社保卡结算系统，在 Oracle 服务器上基于多表连接创建了一个视图，名称为 DIS.CARD_PAYOUT，其结构如下。

- SBJGBH：社保机构编号，VARCHAR2(10)类型。
- SFZHM：身份证号码，VARCHAR2(18)类型。
- KYHZH：银行卡号，VARCHAR2(30)类型。
- JYJE：交易金额，NUMBER 类型。
- YHJYH：银行交易号，VARCHAR2(100)类型。

- ❑ YYBM：医疗机构编码，VARCHAR2(8)类型。
- ❑ FSSJ：交易发生时间，DATE 类型。

在社保卡结算系统开发过程中，除了 DIS.CARD_PAYOUT 视图外，还会用到以下四个表（以下表结构中仅列出了与开发本系统相关的字段）。

（1）医疗机构表 MD.INSTITUTION_NATL。

- ❑ SBJGBH：社保机构编号，VARCHAR2(8)类型，医院所属的社保机构。
- ❑ YYBM：医院编码，ARCHAR2(18)类型。
- ❑ YYCM：医院名称，VARCHAR2(50)类型。

（2）参保人员信息表 MD.EMP_NATL。

- ❑ GRBH：个人编号，即身份证号码，VARCHAR2(18)类型。
- ❑ XM：人员姓名，VARCHAR2(20)类型。

（3）社保卡账号表 MD.CARD_ACCOUNT。

- ❑ GRBH：个人编号，即身份证号码，VARCHAR2(18)类型，参保人员所属社保机构。
- ❑ SBJGBH：社保机构编号，VARCHAR2(8)类型。
- ❑ KYHZH：社保卡号，即银行卡号，VARCHAR2(20)类型。

（4）社保机构表 SI.SI_NATL。

- ❑ SBJGBH：社保机构编号，VARCHAR2(20)类型。
- ❑ SBJGMC：社保机构名称，VARCHAR2(100)类型。

为实现社保卡结算系统的功能，在社保管理系统数据库中需要添加以下几个表。

（1）医疗机构对应表 MD.INSTITUTE_DICTIONARY。

该表保存了银行数据库的医疗机构编码与社保管理系统数据库医疗机构编码之间的对应关系。在银行数据库中商户号和终端号对应某个医疗机构，而在社保管理系统中医院编码和社保机构编号对应于某个医疗机构。同一个医疗机构在银行数据库的代码和社保管理系统数据库的代码不同，为了实现银行数据与社保数据的交流，需要确定这两个不同编码的对应关系。MD.INSTITUTE_DICTIONARY 表保存了这个对应关系，表中各字段如下。

- ❑ CORNO：商户号，VARCHAR2(30)类型。
- ❑ TERM：终端号，VARCHAR2(20)类型。
- ❑ YYBM：医院编码，VARCHAR2(20)类型。
- ❑ SBJGBH：社保机构编号，VARCHAR2(8)类型。

（2）社保机构所属区县表 MD.SBJGQX。

该表描述了社保机构基本信息，包括编码、名称和所属区县，主要用于按照区县进行统计报表。表结构如下。

- ❑ SBJGBH：社保机构编号，VARCHAR2(8)类型。
- ❑ SBJGMC：社保机构名称，VARCHAR2(100)类型。
- ❑ SSQX：所属区县，VARCHAR2(50)类型。

（3）银行卡消费明细表 MD.BANK_TRANSACTION。

该表包含了从银行 dbf 文件导入的银行卡消费数据，表结构如下。

- ❑ ID：记录编号，CHAR（20）类型，主键。
- ❑ WordDate：交易日期，CHAR(10)类型。
- ❑ Corno：商户号，VARCHAR2(30)类型，即医院或者药店在银行的客户编号。

- Name：商户名称，VARCHAR2(60)类型，即医院或者药店名称。
- Accno：银行卡号，VARCHAR2(40)类型，即社保卡号。
- Amt：消费金额，NUMBER 类型。
- Fee：银联交易手续费，NUMBER 类型，按照消费金额的一定比例收取手续费。
- Realamt：实际结算金额，NUMBER 类型，社会保障局将按照这个金额与医疗机构进行结算。
- Type：消费类型，VARCHAR2(10)类型，为 4 个值之一：PCA 正常，PPT 退货，PCC 消费冲正，PPC 退货冲正。
- Term：终端机号，VARCHAR2(20)类型，银联刷卡的终端机号。
- Time：消费时间，DATE 类型，此笔消费所发生的具体时间。
- Cserno：银联交易流水号，VARCHAR2(20)类型。
- Audio_No：审核编号，CHAR(10)类型。

（4）银行卡消费审核表 MD.BANK_TRANSACTION_AUDIT。

该表包含了对银行卡消费数据的审核情况。表结构如下。

- AUDIT_NO：审核编号，CHAR(10)类型，主键。
- AUDIT_DATE：审核日期，DATE 类型，默认值为当前日期。
- AUDIT_USER：审核人，VARCHAR2(10)类型。
- YYBM：医院编码，VARCHAR2(20)类型。
- SBJGBH：社保机构编号，VARCHAR2(10)类型。
- YYMC：医院名称，VARCHAR2(50)类型。
- AMT_SUM：总消费金额，NUMBER 类型。
- FEE_SUM：总手续费，NUMBER 类型。
- REALAMT_SUM：总的实际结算金额，NUMBER 类型。
- MONEY_PAYED：已经向医疗机构支付的金额，NUMBER 类型。
- CANCEL：取消审核标志，CHAR(1)类型，0 表示未取消，1 表示已经取消。

（5）医院权限表 MD.USER_HOSPITAL_RIGHT。

该表描述了用户对医院数据的权限信息，即某个用户是否有权查看某医院的数据。表结构如下。

- USER_ID：用户 ID，VARCHAR2(10)类型。
- YYBM：医院编码，VARCHAR2(18)类型。
- SBJGBH：社保机构编码，VARCHAR2(8)类型。
- RIGHT：权限，CHAR(1)类型，1 表示有权，否则无权。

（6）数据上传日志表 MD.UPLOAD_LOG。

该表记录了银行数据上传情况，每当用户将银行提供的 dbf 格式的社保卡消费明细上传时，上传信息就会记录到这个表中。表结构如下。

- UPLOAD_ID：上传 ID，NUMBER 类型，自动增长，主键。
- USER_ID：上传用户 ID，VARCHAR2(10)类型。
- UPLOAD_DATE：上传日期，DATE 类型，默认值为当前日期。
- TOTAL_RECORD：上传文件中的总记录数，NUMBER 类型。
- CORRECT_RECORD：上传文件中成功导入的记录数，NUMBER 类型。

- ERROR_RECORD：上传文件中导入失败的记录数，NUMBER 类型

数据库中还包括权限管理相关的几个表，表结构在第 11 章中已经介绍过，此处不再赘述。

13.1.3　项目框架

社保卡结算系统需要运行在现有的服务器上，项目开发所采用的技术必须与服务器兼容。由于软件开发时服务器上仅具备.NET Framework 2.0 运行环境，而不具备.NET Framework 4.0 运行环境，且服务器上应该尽可能少的安装非必需的软件，所以本项目基于.NET Framework 2.0 进行开发。与框架版本相对应，数据访问不能使用 LINQ 和 Entity Framework，而是使用 Enterprise Library。

本项目采用多层结构设计，使用实体框架完成数据访问。解决方案中应该包含以下几个项目：业务实体层、数据访问层、业务逻辑层、Web 表现层、单元测试项目。搭建整个解决方案框架的步骤如下。

（1）创建一个空白解决方案 AccountCheck。

（2）在解决方案中添加一个类库项目 Account.Entity 作为业务实体层。

（3）在解决方案中添加一个类库项目 AccountCheckDal 作为数据访问层。

（4）在解决方案中添加一个类库项目 AccountCheckBll 作为业务逻辑层。

（5）在解决方案中添加一个 ASP.NET Web 应用程序项目 AccountCheckWeb 作为表现层。

（6）在解决方案中添加一个类库项目 DalTest 作为数据访问层的单元测试项目。

（7）在各个项目之间添加引用关系，表现层、业务逻辑层、数据访问层都引用业务实体层，业务逻辑层引用数据访问层，表现层引用业务逻辑层，测试项目引用被测试项目。整个解决方案结构如图 13.1 所示。



图 13.1　社保卡结算系统项目结构

13.2　Oracle 数据库简介

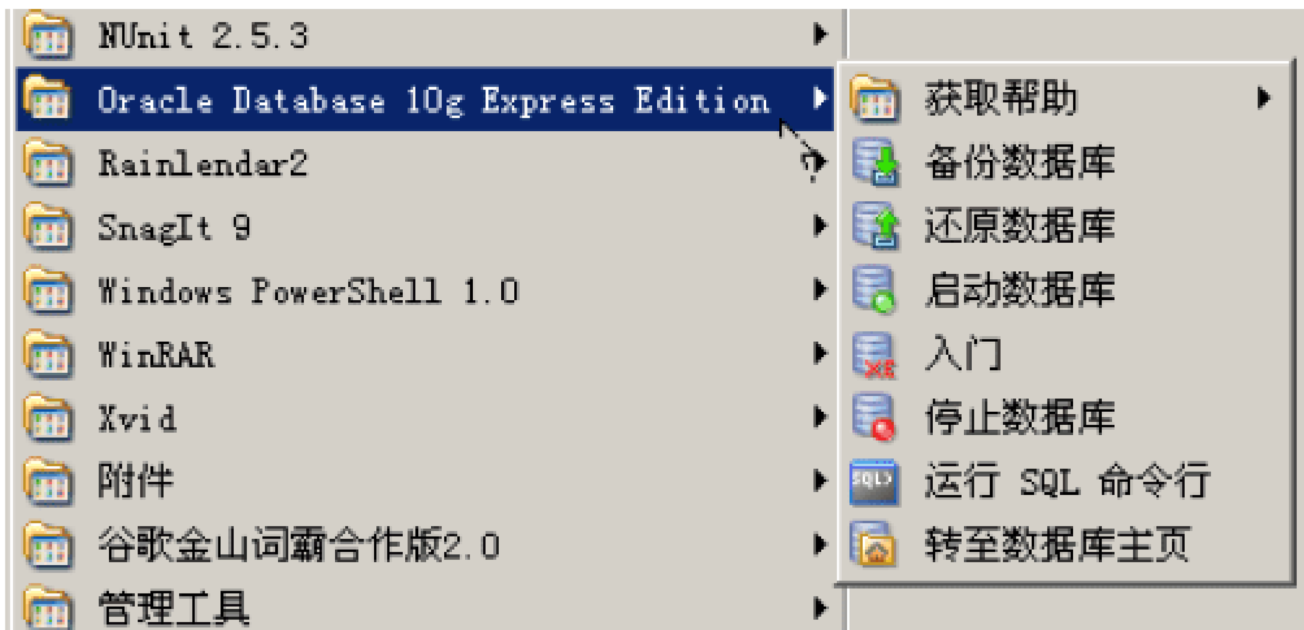
A 市人力资源和社会保障局的社保管理系统采用 Oracle 作为数据库，本章所开发的社保卡结算系统需要使用此数据库中的多个表。为了便于数据共享和管理，本项目直接使用社保管理系统已有的 Oracle 数据库作为后台数据库，并根据需要在其中创建表。

对于大多数 ASP.NET 程序员来说，所使用的后台数据库都是 SQL Server 数据库，国内的 ASP.NET 开发书籍也大都以 SQL Server 数据库为例讲解数据访问功能，因此有必要在本项目正式开发以前对 Oracle 数据库进行简单介绍。

13.2.1　安装 Oracle

Oracle 数据库安装程序可以从 Oracle 官方网站免费下载，下载地址为

http://www.oracle.com/technology/software/index.html。Oracle 数据库有多个版本，如企业版（Enterprise Edition）、标准版（Standard Edition）、速成版（Express Edition）等。其中企业版功能最强大，但是占用资源多，而速成版功能较简单，占用资源少，且为免费软件。速成版在使用上还包括其他一些限制，如仅支持单处理器，最多只能管理 4G 数据和 1G 内存。对于较小规模的应用程序来说，Oracle 速成版能够满足程序对数据库的要求。



13.2　OracleXE 程序菜单

由于本项目在开发过程中仅用到最基本的 Oracle 数据库功能，所以笔者在开发过程中使用的是 OracleXE（即速成版）。在下载 OracleXE 时要注意选择多国语言版，这样才能支持中文界面。Oracle 的安装与普通软件安装类似，按照安装向导的提示一步一步进行操作即可。安装完成后，Windows 开始菜单中将出现以下内容，如图 13.2 所示。

13.2.2　管理用户

Oracle 数据库通常需要创建多个用户，每个用户拥有不同的数据库资源和权限，其中系统用户 sys 具有最高权限。OracleXE 提供一个 Web 方式的图形化界面管理工具，这一点与 SQL Server 有很大区别。从 Windows 开始菜单中选择 Oracle Database 10g Express Edition|“转到数据库首页”命令，则会打开如图 13.3 所示的数据库登录界面。在其中输入用户名和密码（其中密码为安装 OracleXE 时设置的密码），然后单击“登录”按钮。

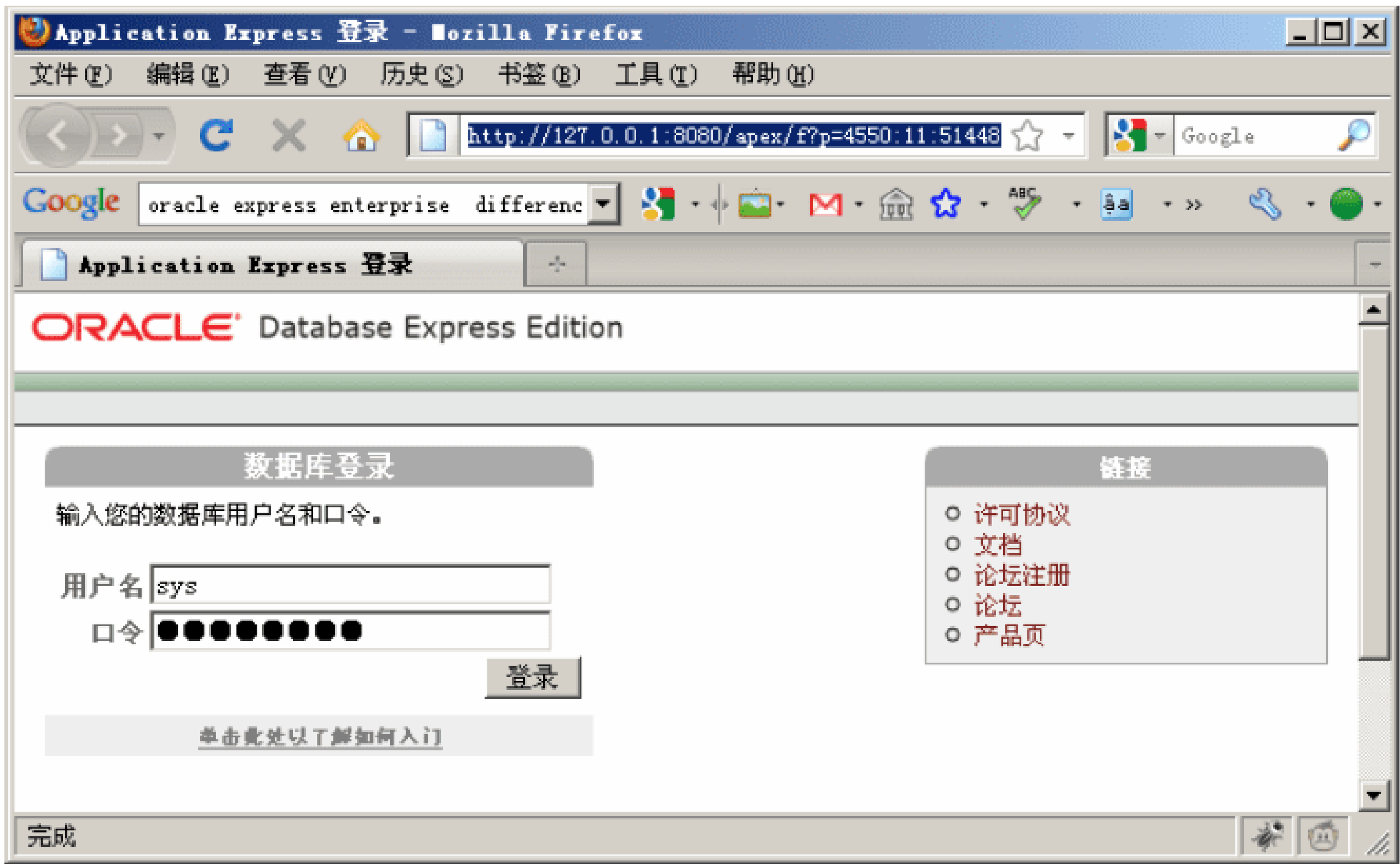


图 13.3　Oracle 管理程序登录界面

登录 Oracle 成功后，会跳转到如图 13.4 所示的 Oracle 数据库管理主页，其中显示了常用的管理工具按钮，在页面右侧显示了帮助链接和数据库性能监视器。

在 Oracle 数据库管理主页，依次选择“管理”|“数据库用户”|“创建用户”命令，则进入创建数据库用户页面，如图 13.5 所示。在其中可以输入要创建的用户名和密码，并为用户分配适当的权限。

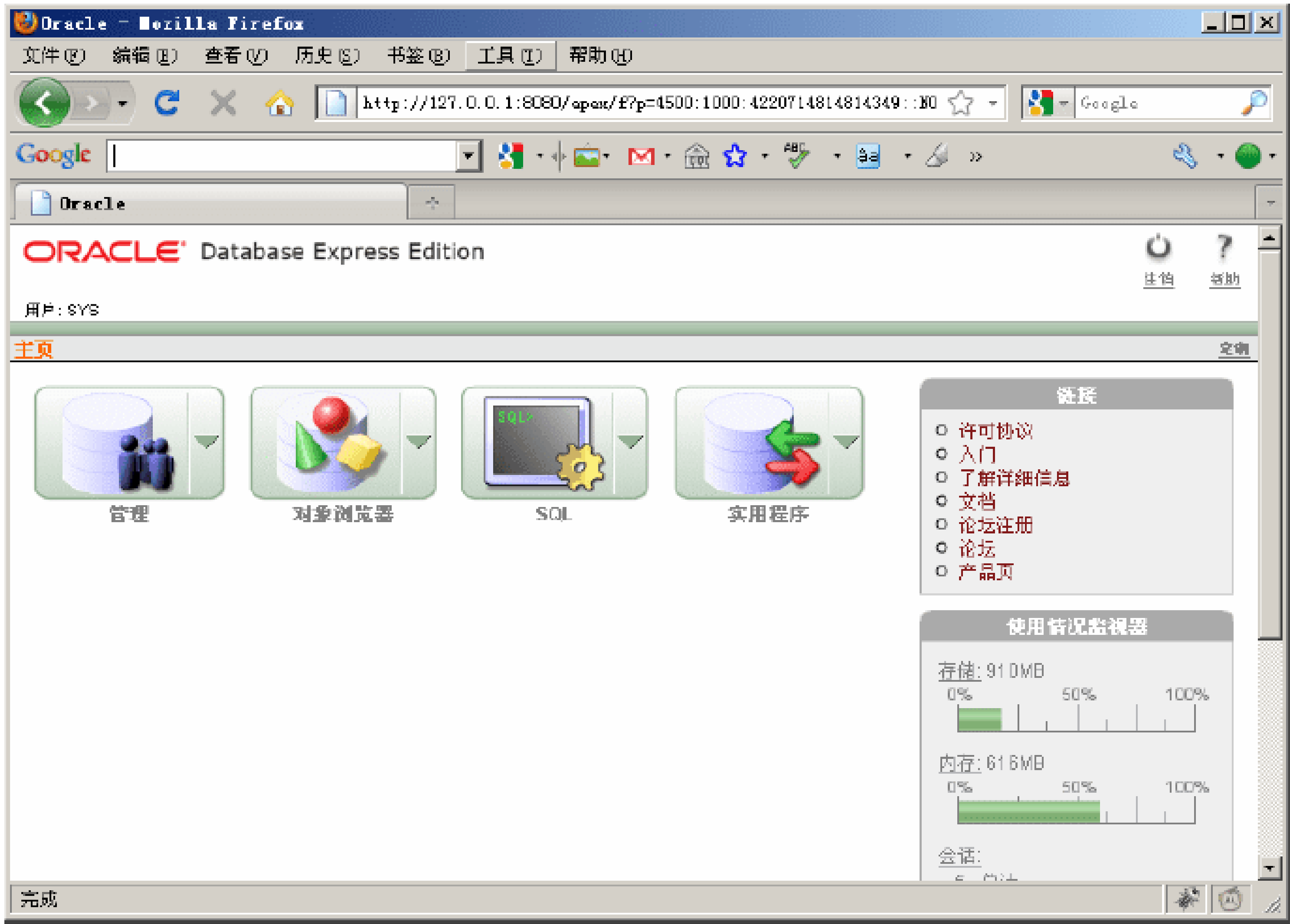


图 13.4　Oracle 数据库管理主页

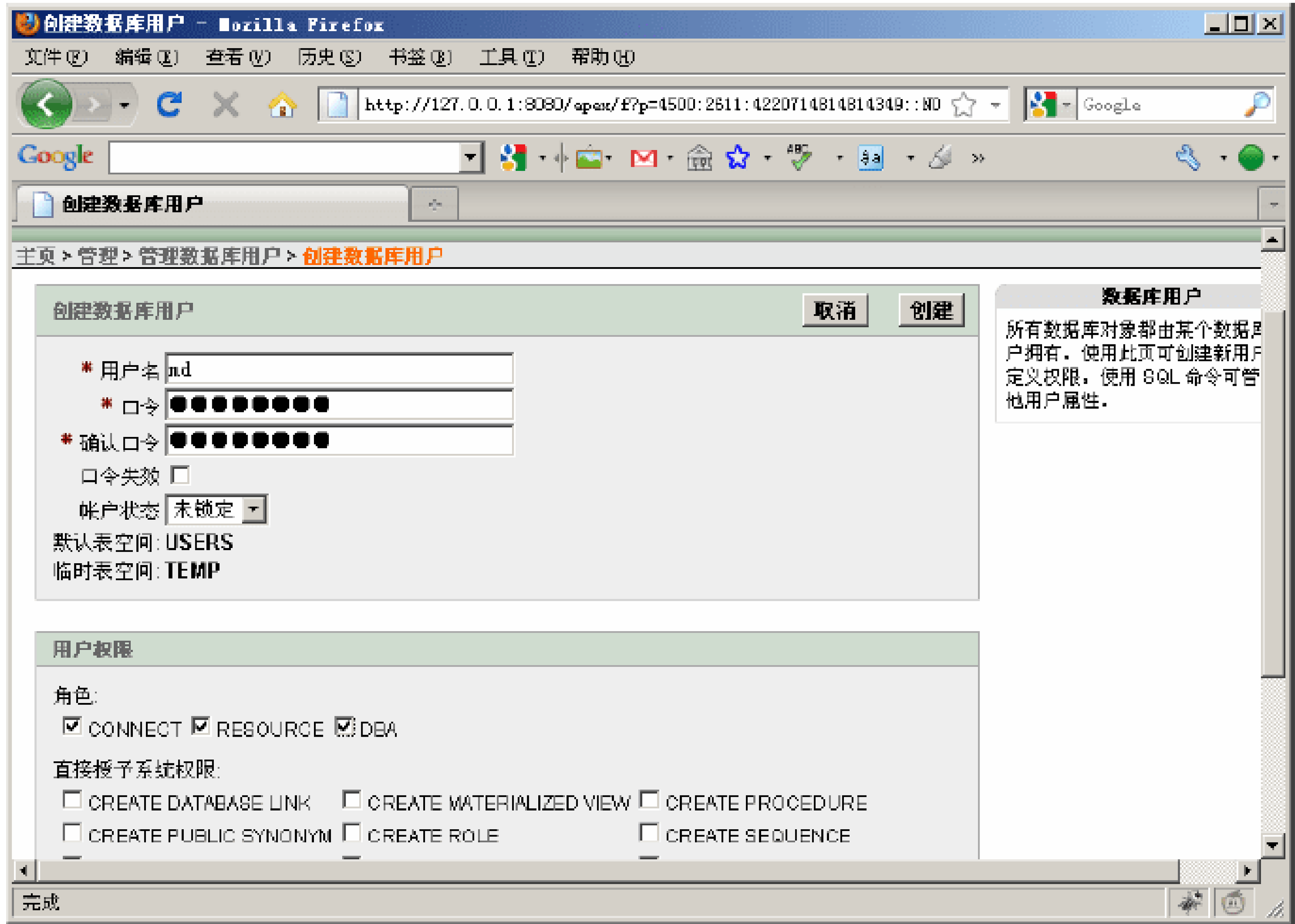


图 13.5　创建数据库用户

在 Oracle 中创建一个用户后，会自动创建一个与用户名相同的方案（Schema）。简单来说，方案就是一个数据库对象（如表、视图等）的容器，一个数据库对象（如表）完整名称由方案名与对象名（如表名）组成，如在前面内容中提到的 MD.CARD_ACCOUNT 表示 MD 方案下的 CARD_ACCOUNT 表。

在社保卡结算系统中，需要用到 3 个数据库用户和方案，分别是 MD、DIS 和 SI。在 Oracle 数据库管理工具中添加这 3 个用户，并为其赋予 DBA 的用户权限。

13.2.3　管理表和数据

在 Oracle 中添加表、修改表结构等操作与 SQL Server 不同，下面将介绍如何在 Oracle 中对表进行管理。在 Oracle 数据库管理主页中，依次选择“对象浏览器”|“浏览”|“表”命令，则进入对象浏览器页面。页面左侧列出了当前用户的所有表，从左侧选择一个表，则在右侧此表的详细信息以及相关的操作按钮，如图 13.6 所示。

提示： 如果在对象浏览器中找不到某个表，则可能是由于数据库登录用户不正确。例如，图 13.6 所示为 MD 用户登录后对象浏览器页面，其中列出了 MD 下的所有表，却不显示 DIS、SI 等其他用户的表。

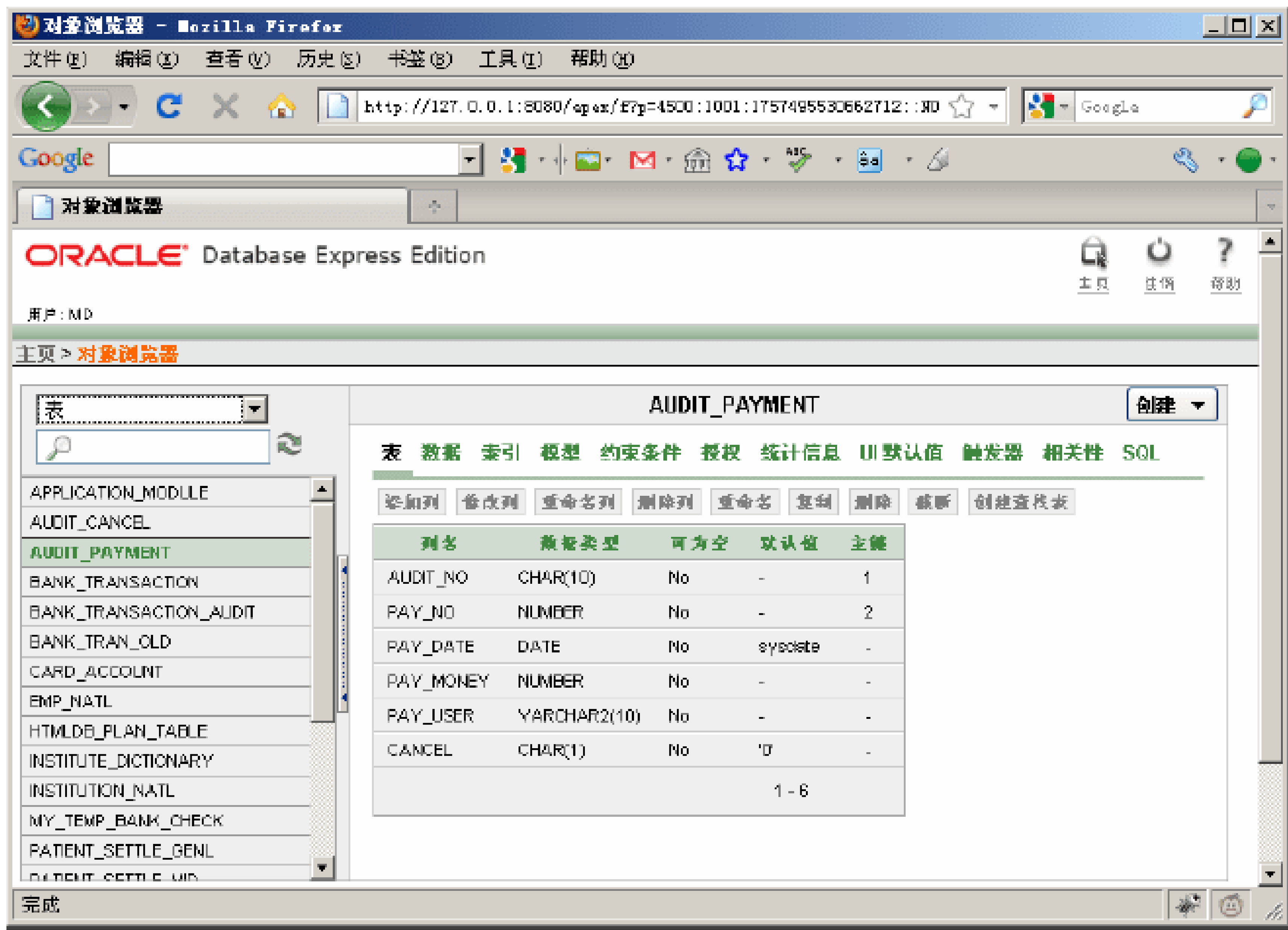


图 13.6　对象浏览器

在图 13.6 所示的对象浏览器页面中，有“添加列”、“修改列”、“删除列”、“删除”等多个功能按钮，可以选择这些按钮以执行相应操作。如果要查看其他类型的数据库对象（如存储过程等），则从对象浏览器页面左侧顶部的下拉列表中选择相应类型即可。

如果要创建新表，则从对象 13.6 所示页面中单击“创建”按钮，在接下来的选择创建对象类型页面中选择创建表，则进入如图 13.7 所示的创建表页面，在其中输入表名、各个字段名称和类型。

提示： Oracle 数据库与 SQL 数据库的数据类型有较大不同，Oracle 的 varchar2 类型相当于 SQL Server 的 varchar 类型，Oracle 的 date 类型相当 SQL Server 的 datetime 类型，Oracle 的 number 类型既可以表示整数也可以表示小数。

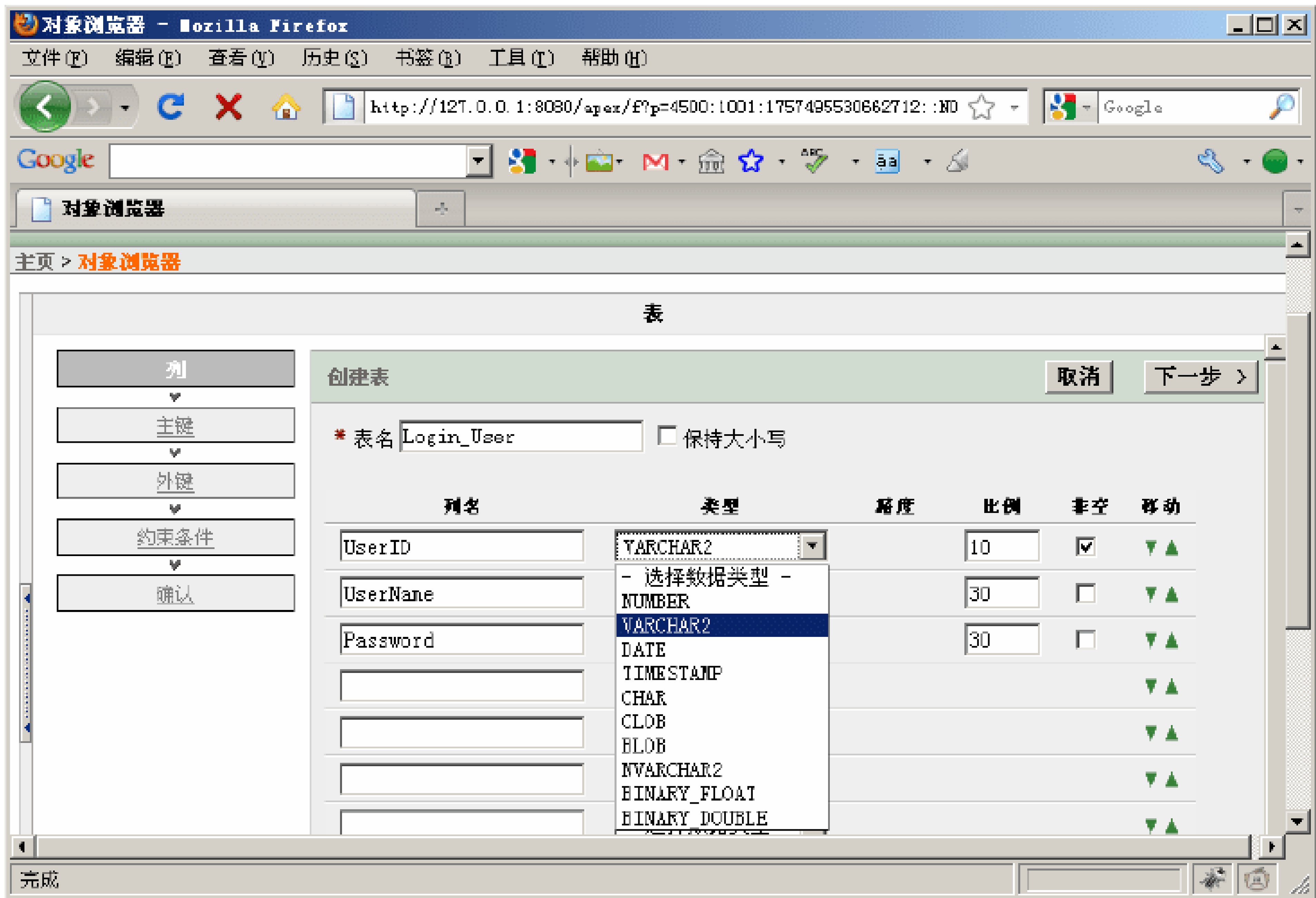


图 13.7　创建新表

在图 13.7 所示页面中填写各个字段后，单击“下一步”按钮，则进入设置主键页面，如图 13.8 所示。在 Oracle 数据库中为表设置主键时，有一个序列填充的选项。这个选项类似于 SQL Server 数据库中的自动增长列，可以使用一个指定的序列自动为新插入的数据设置主键。如果不需要填充主键，则选择“未填充”选项。

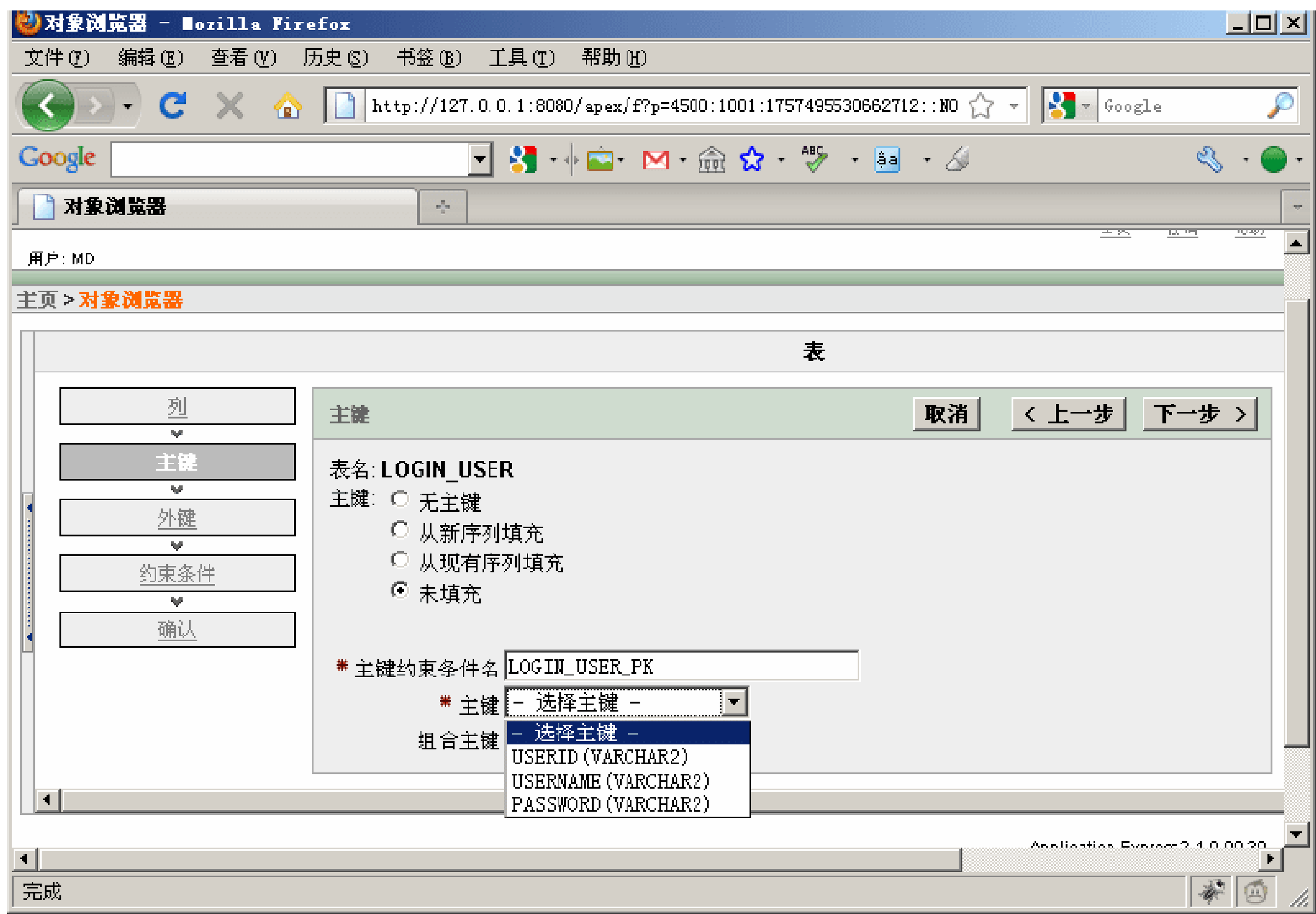


图 13.8　设置主键

为表选择一个或多个字段作为主键后，一直单击“下一步”按钮，直到出现“完成”按钮。单击“完成”按钮，则进入到确认页面，其中列出了要创建表的详细信息，如图 13.9 所示。在其中单击“创建”按钮，就最终完成了表的创建。

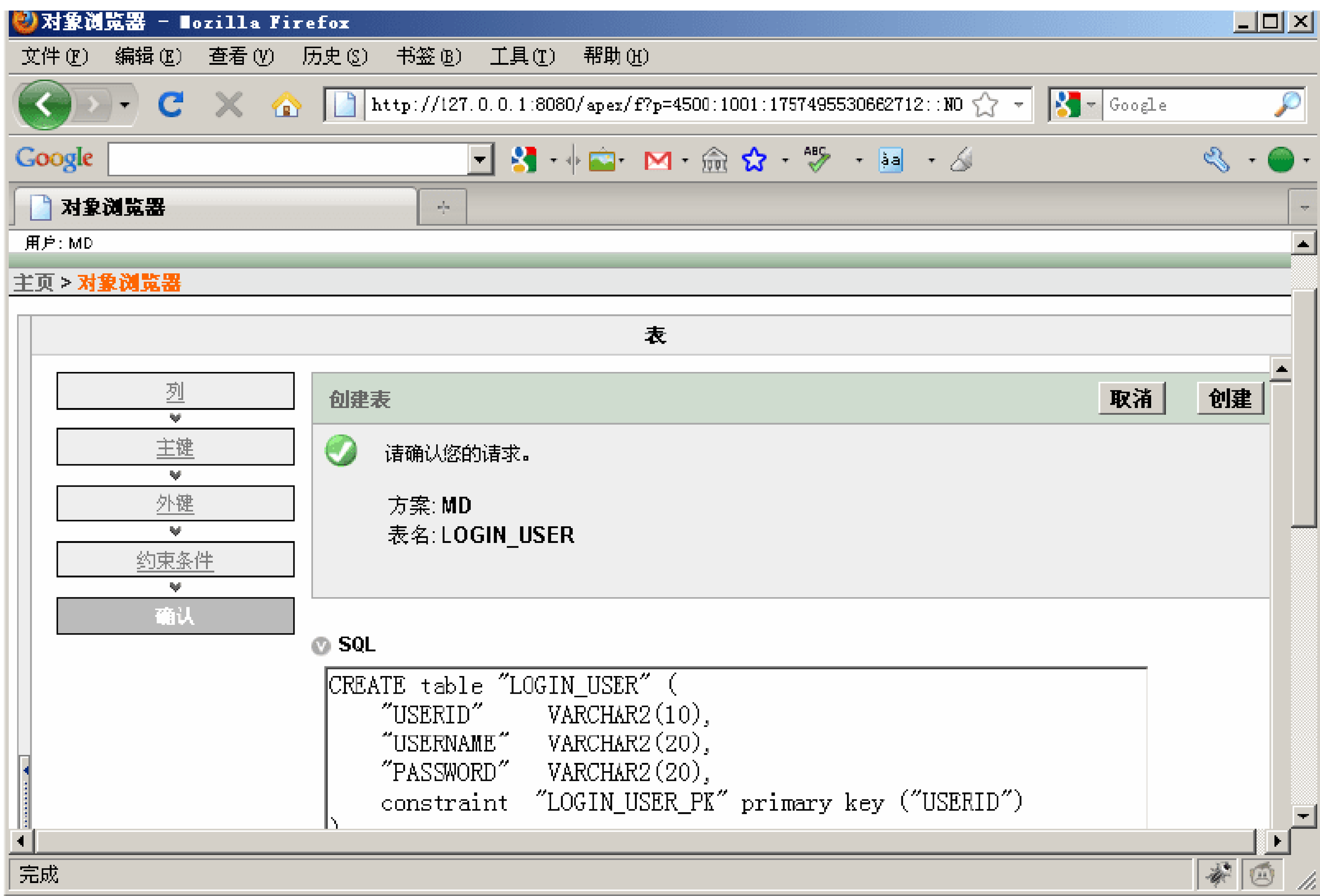


图 13.9　创建表确认页面

如果要为表添加数据，则在图 13.6 所示的对象浏览器中选中一个表，然后单击“数据”按钮，再单击“插入行”按钮，则进入如图 13.10 所示页面，在其中可以向表中添加数据。

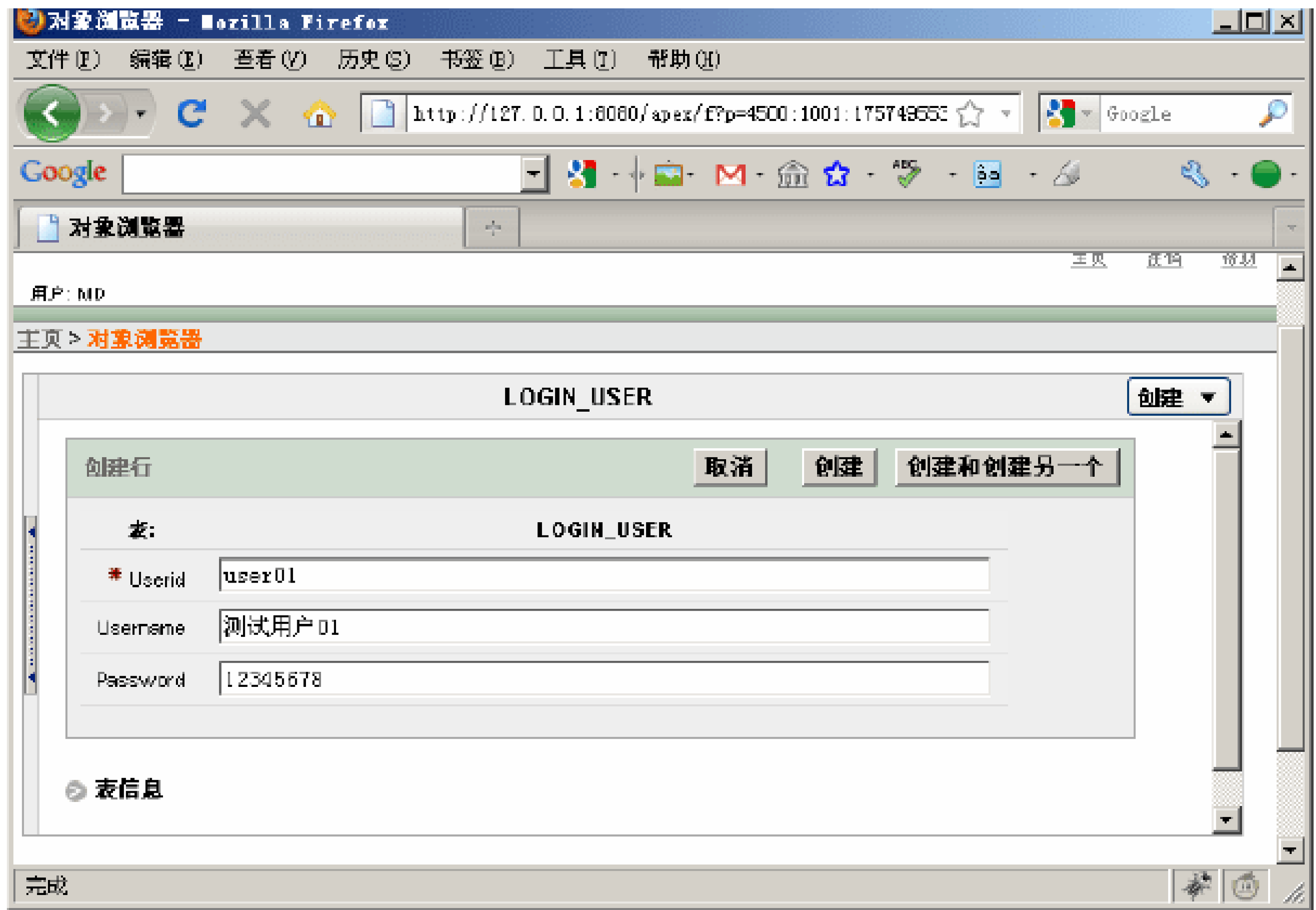


图 13.10　向表中添加数据

如果要在 Oracle 中执行 SQL 命令，从数据库首页上选择“SQL 命令”|“输入命令”，则打开 SQL 命令页面，如图 13.11 所示。由于数据库的数据量通常比较大，Oracle 并不显示全部的查询结果，而是只显示结果的前 N 条记录，这个 N 的数值可以从图 13.11 所示页面的顶部下拉列表中进行设置。

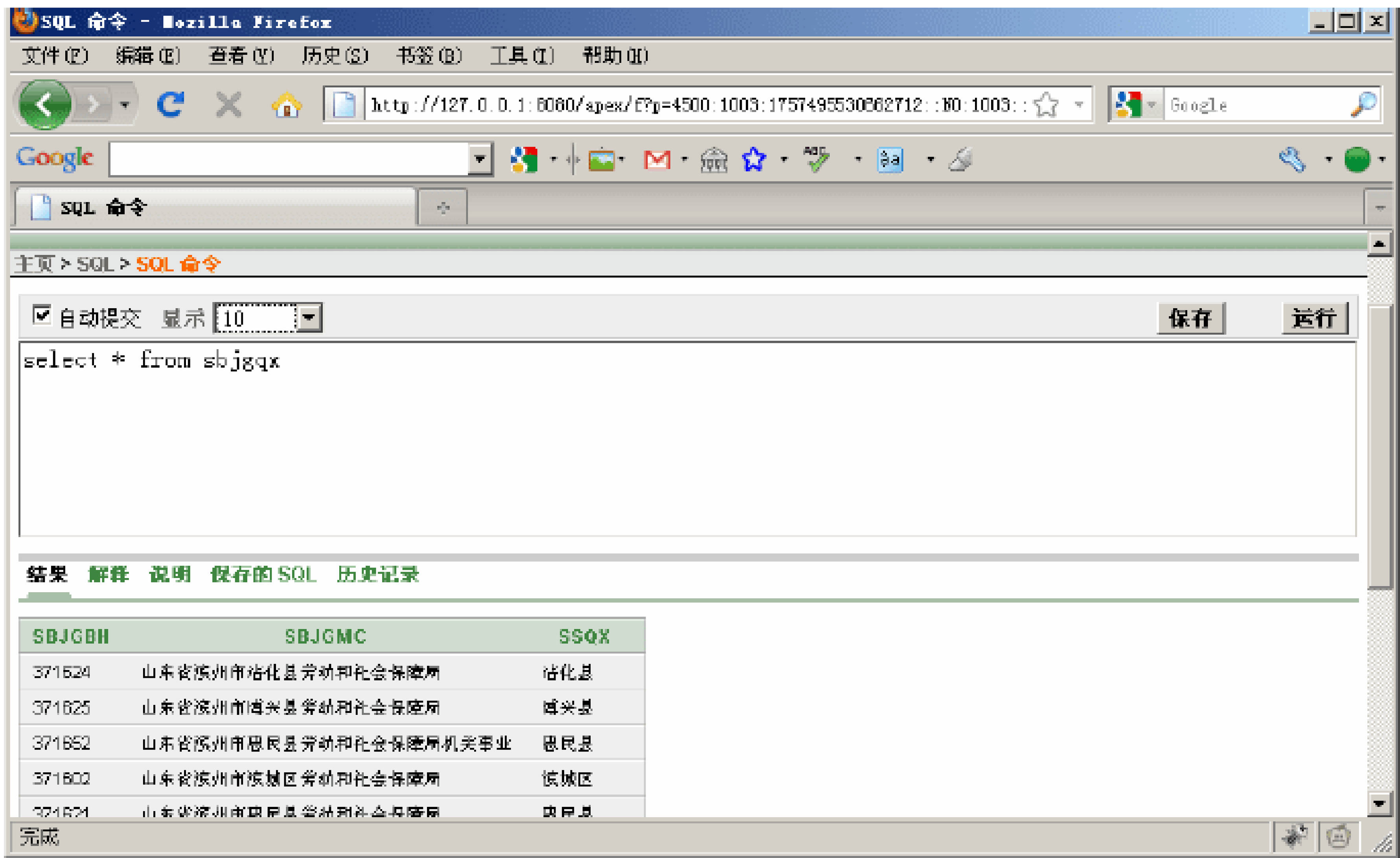


图 13.11　输入和执行 SQL 命令

13.2.4　PL/SQL 简介

Oracle 数据库中使用 PL/SQL 作为数据查询语言，PL/SQL 语言是对标准 SQL 语言的扩展。SQL Server 数据库使用的 SQL 语言称为 T-SQL 语言，也是对标准 SQL 语言的扩展。PL/SQL 语言最基本的增删改查语句与 T-SQL 语言相同，如下面的代码所示。

```
INSERT INTO Web User (ID, Name, Password) VALUES('user10', 'zhang san',
'123456');
UPDATE Web_User SET Name='张三' WHERE ID='user10';
SELECT * FROM Web_User;
DELETE FROM Web_User WHERE ID='user10';
```

PL/SQL 与 T-SQL 有一定差别，对于只熟悉 SQL Server 数据库的开发人员来说，刚开始使用 Oracle 数据库可能会感到不习惯。下面主要介绍几点 PL/SQL 与 T-SQL 不同之处。

1．查询前N条记录

在 T-SQL 中，使用 TOP 关键字查询前 N 条数据，如下代码所示。

```
SELECT TOP 10 * FROM WEB_USER
```

在 PL/SQL 中不识别 TOP 关键字，要想查询前 N 条数据，需要使用 ROWNUM。ROWNUM 表示一条记录在查询结果集中的行号，从 1 开始。如果要查询前 N 条记录，则应使用以下语句。

```
SELECT * FROM Web_User WHERE ROWNUM<=10
```

需要注意的是，ROWNUM 不支持大于比较。例如，下面的语句查询结果永远为空，即使表中有多条数据。

```
SELECT * FROM Web_User WHERE ROWNUM>1
```

如果确实要查询结果集中某个行以后的数据，则应该使用子查询，如下代码所示。

```
SELECT * FROM
(SELECT ID,NAME,PASSWORD, ROWNUM AS RN FROM Web_User )
WHERE RN>1
```

2．数据分页

由于 Oracle 数据库的 ROWNUM 函数能够标识一条记录在整个查询结果中的行号，所以在 Oracle 数据库中实现数据分页功能较为方便，只需查询 ROWNUM 值在某个范围之内的数据即可。由于 ROWNUM 函数不支持大于比较，所以分页查询时同样需要使用子查询。下面的代码查询出 Web_User 表中的第 2 页数据（每页 10 条数据）。

```
SELECT * FROM
(SELECT ID,NAME,PASSWORD, ROWNUM AS RN FROM Web_User )
WHERE RN>20 and RN<=30
```

3．使用变量

在 PL/SQL 中定义和使用变量的语法与 T-SQL 也不相同，给变量赋值时要使用:=符号，在查询语句中使用变量时要在变量名前面加:前缀。以下代码是一个定义和使用变量的例子。

```
DECLARE                               --必须有此关键字，表示定义变量
var_id varchar2(10);                  --定义变量时变量名前面没有冒号
var_pass varchar2(20);
BEGIN
var_id:='admin';                      --给变量赋值时变量名前面没有冒号
var_pass:='123456';                   --赋值号为:=
--在查询中使用变量时变量名前面有冒号
UPDATE web_user SET password=:var_pass WHERE id=:var_id;
END;
```

提示：在 SQL Server 的 T-SQL 中，变量名以电子邮箱符号@为前缀，如@myvar，而在 Oracle 的 PL/SQL 中，变量名以冒号:为前缀，如:myvar。如果在 PL/SQL 中把变量名写成了以@开头（如@myvar），命令执行时就会出现错误。

4．执行多条语句

在 PL/SQL 中，如果要一次执行多条语句，那么这些语句必须包含在一对 BEGIN 和 END 之内，否则就会出现语法错误。如下例所示。

```
--下面这两条语句如果一起执行就会出现错误
UPDATE web user SET password='123456' WHERE id='admin';
UPDATE web_user SET password='654321' WHERE id='user01';
--下面这两条语句可以一起执行，不会出错
BEGIN
UPDATE web user SET password='123456' WHERE id='admin';
UPDATE web user SET password='654321' WHERE id='user01';
END;
```

5．类型转换

在执行数据库查询时，经常需要进行类型转换。限于篇幅，此处仅介绍 PL/SQL 中常用的类型转换函数。TO_CHAR()函数实现将数据以特定格式转换成字符类型，TO_NUMBER()函数将数据转换为数值类型，TO_DATE()函数将数据转换为日期类型。各函数语法如下。

```
TO_CHAR(要转换的数据,格式字符串)
TO_NUMBER(要转换的数据,格式字符串)
TO_DATE(要转换的数据,格式字符串)
```

下面是几个类型转换的例子。

```
SELECT TO CHAR(SYSDATE, 'YYYY年MM月DD日') AS today FROM DUAL;
--上一条语句输出结果为：2010年04月25日
SELECT TO CHAR(SYSDATE, 'DS') AS today FROM DUAL;
--上一条语句输出结果为：2010-04-25
SELECT TO CHAR(SYSDATE, 'HH:MI:SS') AS NOW12, TO CHAR (SYSDATE, 'HH24:MI:SS')
AS NOW24 FROM DUAL;
--上一条语句输出结果为：03:38:18       15:38:18
SELECT TO_CHAR(12345.67,'99,999.99') AS num FROM dual
--上一条语句输出结果为：12,345.67
SELECT TO_DATE('2010-01-01', 'YYYY-MM-DD') AS my_date FROM DUAL;  --字符
转换为日期
SELECT TO_NUMBER('123456.78') AS my_number FROM DUAL;          --字符转换
为数值
```

13.3　母版页设计

社保卡结算系统中大部分页面布局都包含三部分：页头、正文和页脚，其中页头显示工具栏，页脚显示网站说明和联系方式，这两部分内容对于大多数页面来说是相同的，可以放到母版页中。本节将介绍母版页的设计和实现。

13.3.1　Header 用户控件

Header 用户控件位于页面顶端，用于显示程序名称、当前日期、当前用户等提示信息，并以图形按钮方式显示功能导航栏。Header 用户控件外观如图 13.12 所示。

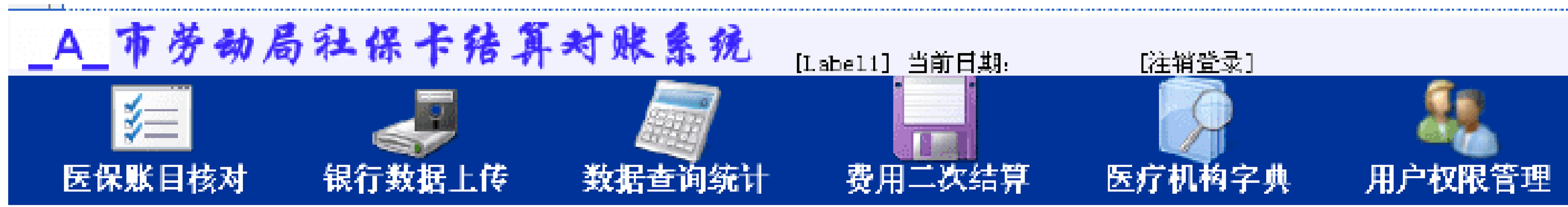


图 13.12　Header 用户控件

（1）在表现层项目 AccountCheckWeb 中添加一个文件夹 UserControls，此文件夹用于存放项目中的用户控件。

（2）在 UserControls 文件夹中添加一个用户控件 Header.ascx，在 Header.ascx 文件中添加以下代码。

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="Header.
ascx.cs" Inherits= "AccountCheckWeb.UserControls.Header" %>
<div style="background:#f2f2fe; width:1000px; text-align:left;">
<div>
<img src="../images/title.png" alt="" />          <!-- LOGO -->
<span style="margin-left:auto; margin-right:30px;">
<asp:Label ID="Label1" runat="server" Text=""></asp:Label>
                                                  <!-- 显示用户名称 -->
当前日期：<%=DateTime.Now.ToLongDateString() +" " +DateTime.Today.
DayOfWeek %> </span>
<span style="margin-left:30px;">
<asp:LinkButton  ID="LinkButton1" runat="server" onclick="LinkButton1_
Click">
[注销登录]</asp:LinkButton></span>
</div>
<!--以下内容显示各个导航菜单-->
<div id="NavMenu">
<div class="ImageMenu" runat="server" id="check">
<img src="../images/check.png" alt="医保账目核对" /><br/>
<a href="../Default/CheckAccount2.aspx">医保账目核对</a></div>
<div class="ImageMenu" runat="server" id="upload">
<img src="../images/upload.png" alt="银行数据上传" /><br/>
<a href="../Default/UploadData.aspx">银行数据上传</a></div>
<div class="ImageMenu" runat="server" id="Div1">
<img src="../images/calculator.png" alt="数据查询统计" /><br/>
<a href="../Report/ConsumeReportPage.aspx">数据查询统计</a></div>
<div class="ImageMenu" runat="server" id="Div2">
<img src="../images/floppy.png" alt="费用二次结算" /><br/>
<a href="../Default/PayAgainPage.aspx">费用二次结算</a></div>
<div class="ImageMenu" runat="server" id="dictionary">
<img src="../images/dictionary.png" alt="定点医疗机构字典" /><br/>
<a href="../Default/InstituteDictionary.aspx">医疗机构字典</a></div>
<div class="ImageMenu" runat="server" id="user">
<img src="../images/users.png" alt="用户权限管理" width="48" /><br/>
<a href="../Default/ManageUser.aspx">用户权限管理</a></div>
</div>
</div>
```

（3）在 Header 用户控件的 Page_Load 事件中，显示当前登录的用户名称，并根据用户类型（医院用户和社保机构用户）设置导航菜单的可见性。医院用户不允许使用“用户字典”、“权限管理”、“数据上传”功能，将这些按钮设置为不可见。

```
protected void Page Load(object sender, EventArgs e)
{
    WebUser user0 = Common.WebUtility.currentUser;     //得到当前登录用户
    bool isHospital = WebUtility.isMedicineShopUser(); //是否医院用户
    Label1.Text = "当前用户：" + user0.name+"  ";
    /* 医院用户与非医院用户（即社保机构用户）具有不同权限
     * 如果是医院用户，则禁用用户字典功能、用户管理功能、数据上传功能 */
    if (isHospital)
    {
        dictionary.Visible = false;
        user.Visible = false;
        upload.Visible = false;
    }
    else
    {
        dictionary.Visible = true;
        user.Visible = true;
        upload.Visible = true;
    }
}
```

（4）在“注销登录”按钮的 Click 事件中，注销当前用户，并跳转到登录页面。

```
protected void LinkButton1 Click(object sender, EventArgs e)
{
    Common.WebUtility.currentUser = null;       //清除 Session 中保存的用户
    Response.Redirect("../login.aspx");         //跳转到登录页面
}
```

13.3.2　Footer 用户控件

Footer 用户控件用于显示用户单位（即 A 市人力资源与社会保障局）地址、联系电话和版权信息。Footer 用户控件代码如下：

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="Footer.
ascx.cs" Inherits="AccountCheckWeb.UserControls.Footer" %>
<div  style="width:1000px;  height:20px;  font-size:12px;  color:#333;
text-align:center;
    border:dashed 1px silver; background:#f2f2f2; float:none; clear:both;">
主办单位：A 市劳动和社会保障局。地址:A 市府右街 37 号。联系电话:12333(传
真)0543-3162837。
<br />技术支持：<a href="mailto:sun.j.l.studio@gmail.com"> sun.j.l.studio-
@gmail. com </a></div>
```

13.3.3　母版页

社保卡结算系统母版页主要包括 3 部分内容：顶部的 Header 控件、底部的 Footer 控

件和页面中间的内容占位符控件。创建母版页的步骤如下。

（1）在 Default 文件夹中添加一个母版页 Nrcm.master。

（2）拖动一个 Header.ascx 用户控件放在母版页顶部。

（3）拖动一个 Footer.ascx 用户控件放在母版页底部。

（4）在母版页中定义浏览器标题，并导入 CSS。Nrcm.master 代码如下：

```
<head runat="server">
    <title>A 市劳动局社保卡结算对账系统</title>
    <link href="../css/NrcmCommon.css" type="text/css" rel="Stylesheet" />
    <asp:ContentPlaceHolder ID="head" runat="server">
     <!--页面头部内容占位符-->
    </asp:ContentPlaceHolder>
</head>
<body><div id="main">
    <form id="form1" runat="server">
    <div>
        <uc1:Header ID="Header1" runat="server" />
    </div>
    <div>
        <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
        <!--页面中间内容占位符-->
        </asp:ContentPlaceHolder>
      </div>
      <div>
        <uc2:Footer ID="Footer1" runat="server" />
    </div>
    </form>
    </div>
</body>
```

母版页外观如图 13.13 所示。

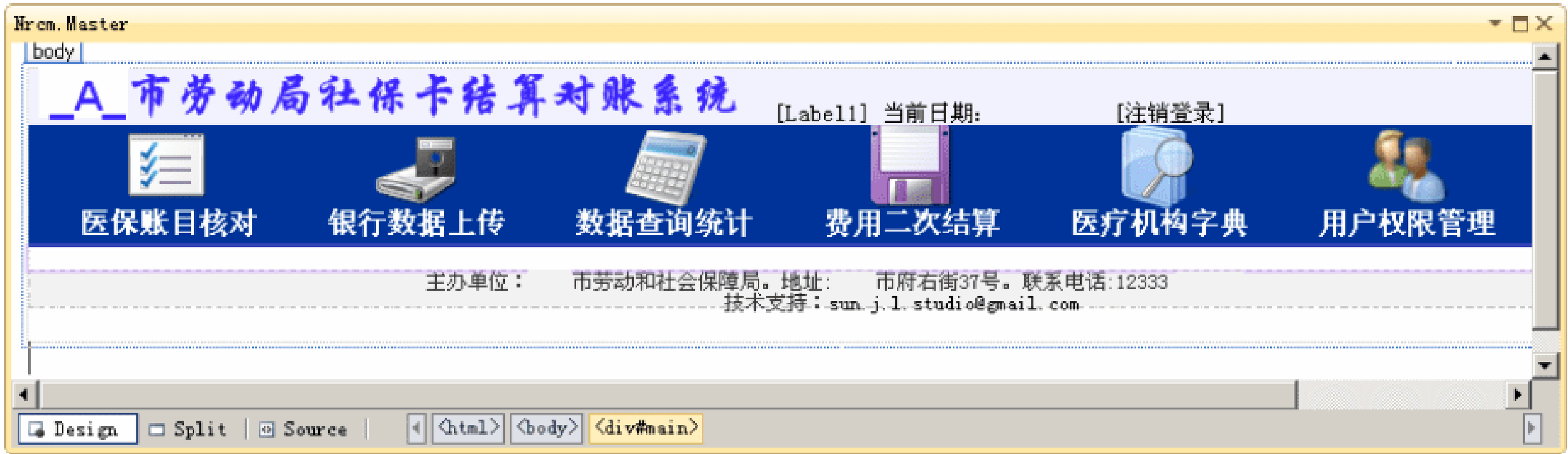


图 13.13　社保卡结算系统母版页

13.4　权限管理

社保卡结算系统的权限管理模块主要功能和设计思路与第 11 章所介绍的基本相同，但是在此基础上，又有一些个性化的要求。这里本系统使用企业库（Enterprise Library）实现数据访问功能，而在第 11 章中使用实体框架（Entity Framework）实现数据访问功能，所以二者的实体类和数据访问层代码有较大不同。

13.4.1　用户和权限管理概述

社保卡结算系统有两类用户：社保机构和医疗机构。社保机构是指 A 市人力资源和社会保障局及其下属机构，医疗机构是指各级医院和药店。其中社保机构的用户和权限管理方式与第 11 章所介绍的方法完全相同，此处不再赘述。但是对于医疗机构来说，这种权限管理方式不太合适。首先，医疗机构众多，如果为每个医疗机构的工人人员都在系统中创建登录用户名，则工作量太大。另外，所有医疗机构的权限完全相同且固定不变，其权限仅限于查询自己医院（或药店）的消费数据。

由于医疗机构众多，而且医疗机构的权限是固定的，A 市人力资源和社会保障局的用户要求不需要给医疗机构创建用户和角色，而是让医疗机构使用各自的医院编码进行登录。各个医疗机构的数据已经在社保管理系统数据库的 MD.INSTITUTION_NATL 表中存在，医疗机构可以使用医院编码字段 YYBM 的内容作为用户名，社保机构编号字段 SBJGBH 的内容作为密码进行登录。根据上述需求，在登录界面中就需要根据两种不同的用户类型进行判断。

前面所述的权限管理的对象都是基于功能模块（或者页面），在社保卡结算系统中还需要另外一种权限管理方式，即基于医疗机构的权限管理。具体来说，对于不同的用户和角色，他们都有权限访问某个模块（如查询医疗机构消费明细等），但是各个用户所允许访问的医疗机构是不同的，例如用户 A 可能允许访问所有医疗机构的数据，用户 B 只允许访问其中 5 个医疗机构的数据。系统管理员可以指定某个用户是否可以访问某医院的数据。

这种权限控制与前述的基于页面访问的权限控制不同，不同用户都可以访问同一个页面，而且可以使用同一页面上所有控件（如单击按钮以执行相应功能），但是他们看到的数据是不相同的。要实现这种权限管理，必须使用一种不同于前述权限管理思路的方式，具体实现见本节后续内容。

13.4.2　数据访问辅助类

社保卡结算系统使用 Enterprise Library 实现数据访问功能，Enterprise Library 中的类如 Database 等提供了大多数本项目所需要的功能。项目中还包含一个自定义的数据访问辅助类 MyUtility，其中封装了一些常用的数据访问操作。

```
//通用数据访问类
internal static class MyDbUtility
{
    internal static Database oracle = DatabaseFactory.CreateDatabase
    ("oracle");//数据库对象
    /// <summary>
    /// 得到查询语句返回的记录总数
    /// </summary>
    /// <param name="sql">查询命令文本</param>
    /// <returns>记录总数</returns>
    internal static int getCount(string sql)
    {
        Database oracle = DatabaseFactory.CreateDatabase("oracle");
```

```
        string temp = " select count(*) from (" + sql + ")";
        DbCommand command = oracle.GetSqlStringCommand( temp);
        object o = oracle.ExecuteScalar(command);
        command.Dispose();
        return Convert.ToInt32(o);
    }
    /// <summary>
    /// 得到查询命令返回的记录总数
    /// </summary>
    /// <param name="command">数据库命令</param>
    /// <returns>记录总数</returns>
    internal static int getCount(DbCommand command)
    {
        return getCount(command.CommandText);
    }
    /// <summary>
    /// 检查表是否在数据库中存在
    /// </summary>
    /// <param name="name">要检查的表名</param>
    /// <returns>是否存在</returns>
    internal static bool checkTableExists(string name)
    {
        string check = "select count(*) from user_tables where table_name=
        upper ('"+name+"')";
        Database oracle = DatabaseFactory.CreateDatabase("oracle");
        DbCommand command = oracle.GetSqlStringCommand(check);
        return Convert.ToInt32(oracle.ExecuteScalar(command)) > 0;
    }
    /// <summary>
    /// 删除指定表（如果表存在）
    /// </summary>
    /// <param name="name">表名</param>
    /// <returns>删除表返回 1，否则返回 0</returns>
    internal static int dropTableIfExists(string name)
    {
        if(checkTableExists(name))
        {
            Database oracle = DatabaseFactory.CreateDatabase("oracle");
            oracle.ExecuteNonQuery(CommandType.Text,"drop table "+name);
            return 1;
        }
        return 0;
    }
}
```

为了尽可能提高 MyDbUtility 类的正确性，及时发现存在的错误，应编写一个单元测试类对 MyDbUtility 类进行测试，测试类代码如下：

```
[TestClass()]
public class MyDbUtilityTest
{
    [TestMethod()]
    public void checkTableExistsTest()
    {
        string name = "BANK TRANSACTION";              //存在的表名
        bool actual;
        actual = MyDbUtility.checkTableExists(name);
```

```
        Assert.AreEqual(true, actual);
        name="NoSuchTable";                             //不存在的表名
        actual=MyDbUtility.checkTableExists(name);
        Assert.AreEqual(false, actual);
    }

    [TestMethod()]
    public void getCountTest()
    {
        string sql = " select * from md.Web_User where id like 'user%'";
                                                        //查找多条记录
        int actual;
        actual = MyDbUtility.getCount(sql);
        Assert.AreEqual(3, actual);
        sql = "select * from md.Web_User where id='admin'";
                                                        //查找一条记录
        actual = MyDbUtility.getCount(sql);
        Assert.AreEqual(1, actual);
        sql = "select * from md.Web_User where id='nosuchuser'";
                                                        //查找不存在记录
        actual = MyDbUtility.getCount(sql);
        Assert.AreEqual(0, actual);
    }
}
```

13.4.3　角色管理

角色管理模块可以浏览、添加、删除、修改角色数据，但是系统管理员角色不允许被删除和修改。实现角色管理模块的具体步骤如下。

（1）在实体类项目 Account.Entity 中添加一个类 UserRole，以描述角色信息。

```
public class UserRole
{
    public const string AdminRoleId = "01";//系统管理员角色为 01，不可删除
    private const string HospitalRoleId = "hospital";  //医院角色编号
    public string id { get; set; }
    public string name { get; set; }
    public string description { get; set; }
    #region 医院角色
    private static UserRole hospitalRole=new UserRole() { id = Hospital
    RoleId };
    public static UserRole HospitalRole
    {
        get
        {
            return hospitalRole;
        }
    }
    #endregion
    //判断角色是否系统管理员
    public bool isAdmin()
```

```
    {
        return id == AdminRoleId;
    }
}
```

（2）在数据访问层项目 AccountCheckDAL 中添加一个类 RoleDB，以实现与角色相关的数据访问功能。

```
// 用户角色数据访问类
public static class RoleDB
{
    public static List<UserRole> getAll()          //得到所有角色数据
    {
        string sql = "select ID,NAME,DESCRIPTION FROM User Role";
        var oracle = MyDbUtility.oracle;
        var reader = oracle.ExecuteReader(System.Data.CommandType.Text,
        sql);
        var list = new List<UserRole>();
        while (reader.Read())
        {
            list.Add(fromDataReader(reader));
        }
        reader.Close();
        return list;

    }
    /// <summary>
    /// 根据角色 ID 得到角色数据
    /// </summary>
    /// <param name="id">要查询的角色 ID</param>
    /// <returns>查询到的角色数据</returns>
    public static UserRole getByID(string id)
    {
        string sql = "select ID,NAME,DESCRIPTION FROM User_Role where
        ID=:id";
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":id", DbType.String, id);
        var reader=oracle.ExecuteReader(command);
        UserRole result = null;
        if (reader.Read())
        {
            result = fromDataReader(reader);
        }
        reader.Close();
        return result;
    }
    /// <summary>
    /// 添加角色数据
    /// </summary>
    /// <param name="role">要添加的角色</param>
    /// <returns>添加的行数</returns>
    public static int add(UserRole role)
    {
        string sql = "insert into User Role (ID,NAME,DESCRIPTION) values
        (:id,:name,:description)";
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":id", DbType.String, role.id);
```

```
        oracle.AddInParameter(command, ":name", DbType.String, role.
        name);
        oracle.AddInParameter(command, ":description", DbType.String,
        role. description);
        return oracle.ExecuteNonQuery(command);
    }
    /// <summary>
    /// 更新角色数据
    /// </summary>
    /// <param name="role">要更新的角色</param>
    /// <returns>被更新的行数</returns>
    public static int update(UserRole role)
    {
        string sql = "update User Role set NAME=:name,DESCRIPTION=:descry-
        ption where ID=:id";
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":id", DbType.String, role.id);
        oracle.AddInParameter(command, ":name", DbType.String, role.
        name);
        oracle.AddInParameter(command, ":description", DbType.String,
        role. description);
        return oracle.ExecuteNonQuery(command);
    }
    /// <summary>
    /// 根据角色 ID 删除角色
    /// </summary>
    /// <param name="id">角色 ID</param>
    /// <returns>被删除的行数</returns>
    public static int delete(string id)
    {
        string sql = "delete from User Role where ID=:id";
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":id", DbType.String, id);
        return oracle.ExecuteNonQuery(command);
    }
    /// <summary>
    /// 根据数据读取器的当前记录生成一个 UserRole 对象
    /// </summary>
    /// <param name="reader">数据读取器</param>
    /// <returns>生成的 UserRole 对象</returns>
    private static UserRole fromDataReader(IDataReader reader)
    {
        UserRole role = new UserRole();
        role.description = reader["description"].ToString();
        role.id = reader["id"].ToString();
        role.name = reader["name"].ToString();
        return role;
    }
}
```

（3）在数据访问层测试项目 DalTest 中，为 RoleDB 添加一个单元测试类并运行，以检测和修改 RoleDB 类的错误。

```
[TestClass]
public class RoleDbTest
{
    [TestMethod]
```

```
    public void testAddRole()
    {
        UserRole role = new UserRole() { id = "**", name = "test", description
        = null };
        int n= RoleDB.add(role);
        Assert.AreEqual(1, n);
        role = RoleDB.getByID("**");        //验证数据库中已存在新角色
        Assert.IsNotNull(role);
    }
    [TestMethod]
    public void testDeleteRole()
    {
        int n=RoleDB.delete("**");
        Assert.AreEqual(1, n);
        var role = RoleDB.getByID("**");    //数据库已不存在指定角色
        Assert.IsNull(role);
    }
    [TestMethod]
    public void testGetAll()
    {
        var list = RoleDB.getAll();
        Assert.IsTrue(list.Count > 0);
    }
    [TestMethod]
    public void testGetByID()
    {
        var role = RoleDB.getByID("01");
        Assert.AreEqual("系统管理员", role.name);    //角色存在且数据正确
        role = RoleDB.getByID("norole");
        Assert.IsNull(role);                         //角色不存在
    }
}
```

（4）在业务逻辑层项目 AccountCheckBLL 中添加一个 RoleBLL 类，这个类的主要功能为调用数据访问层的相应方法，代码与第 11 章相同，此处省略。

（5）在表现层项目 AccountCheckWeb 中添加一个页面 UserRolePage.aspx，代码与第 11 章相同，此处省略。

13.4.4　用户管理

用户管理模块可以查询、浏览、添加、删除、修改用户信息。实现用户管理模块的具体步骤如下。

（1）在业务实体项目 Account.Entity 中添加一个类 WebUser 以描述用户信息。

```
public class WebUser
{
    public string id { get; set; }
    public string name { get; set; }
    public string password { get; set; }
    public UserRole role { get; set; }                    //用户所属角色
    public bool isAdmin()                                 //是否管理员
    {
        return this.role.isAdmin();
    }
}
```

（2）在数据访问层项目 AccountCheckDAL 中添加一个类 WebUserDB，实现与用户相关的数据访问功能。

```
public static class WebUserDB
{
    #region 私有字段（列名和表名）
    private const string IdColumn = "ID";
    private const string NameColumn = "Name";
    private const string PasswordColumn = "Password";
    private const string RoleColumn = "Role";
    private const string WebUserTable = "MD.Web_User";
    #endregion
    //得到所有用户
    public static List<WebUser> getAllUsers()
    {
        List<WebUser> list = new List<WebUser>();
        Database oracle = MyDbUtility.oracle;
        //创建 select 语句
        string sql = string.Format("select {0},{1},{2},{3} from {4}",
            IdColumn,NameColumn,PasswordColumn, RoleColumn, WebUser-
            Table);
        DbCommand command = oracle.GetSqlStringCommand(sql);
        //执行命令，得到数据读取器，读取数据
        using (IDataReader reader = oracle.ExecuteReader(command))
        {
            while (reader.Read())
            {
                list.Add(fromDataReader(reader));
            }
        }
        return list;
    }
    /// <summary>
    /// 修改用户（不修改密码）
    /// </summary>
    /// <param name="user">要修改的用户信息</param>
    public static int updateUser(WebUser user)
    {
        //构建 update 语句
        string sql = string.Format("update {0} set {1}=:name,{2}=:role where
        {3}=:id",
            WebUserTable, NameColumn, RoleColumn, IdColumn);
        Database oracle = MyDbUtility.oracle;
        //创建命令，添加参数，并执行命令
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":name", DbType.String, user.
        name);
        oracle.AddInParameter(command, ":id", DbType.String, user.id);
        oracle.AddInParameter(command, ":role", DbType.String, user.role.
        id);
        return oracle.ExecuteNonQuery(command);
    }
    /// <summary>
    /// 修改密码
    /// </summary>
    /// <param name="id">用户 ID</param>
    /// <param name="pass">新密码</param>
    public static int changePassword(string id, string pass)
```

```
    {
        //构建 update 语句
        string sql = string.Format("update {0} set {1}=:pass where {2}=:id",
        WebUserTable,PasswordColumn, IdColumn);
        Database oracle = MyDbUtility.oracle;
        //创建命令，添加参数，并执行命令
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":pass", DbType.String, pass);
        oracle.AddInParameter(command, ":id", DbType.String, id);
        return oracle.ExecuteNonQuery(command);
    }
    /// <summary>
    /// 根据用户 ID 得到用户信息
    /// </summary>
    /// <param name="id">用户 ID</param>
    public static WebUser getUserByID(string id)
    {
        Database oracle = MyDbUtility.oracle;
        //构建 select 语句
        string sql = string.Format("select {0},{1},{2},{3} from {4} where
        {0}=:id",
            IdColumn, NameColumn, PasswordColumn, RoleColumn, WebUser-
            Table);
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":id", DbType.String, id);
        WebUser user = null;
        //执行查询，得到数据读取器，读取数据
        using (IDataReader reader = oracle.ExecuteReader(command))
        {
            if (reader.Read())
            {
                user = fromDataReader(reader);
            }
        }
        return user;
    }
    /// <summary>
    /// 根据 DataReader 当前记录得到一个用户对象
    /// </summary>
    /// <param name="reader">DataReader 对象</param>
    private static WebUser fromDataReader(IDataReader reader)
    {
        WebUser user = new WebUser()
        {
            //根据 DataReader 各个字段设置 WebUser 各个属性
            id = reader["id"].ToString().Trim(),
            name = reader["name"].ToString(),
            password = reader["password"].ToString()
        };
        string temp = reader[3].ToString();
        user.role = RoleDB.getByID(temp);
        return user;
    }
    /// <summary>
    /// 删除用户
    /// </summary>
    /// <param name="id">要删除的用户 ID</param>
    public static int  deleteUser(string id)
    {
```

```
        string sql = string.Format("delete from {0}  where  {1}=:id",
        WebUserTable, IdColumn);
        Database oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":id", DbType.String, id);
        return oracle.ExecuteNonQuery(command);
    }
    /// <summary>
    /// 添加用户（新添加的用户密码默认为 123456）
    /// </summary>
    /// <param name="user">要添加的用户信息</param>
    public static int addUser(WebUser user)
    {
        //return addUser(user.ID, user.Name, user.Password);
        string sql = string.Format("insert into {0} ({1},{2},{3},{4}) values
        (:id,:name,'123456',:role)",
            WebUserTable, IdColumn, NameColumn, PasswordColumn, RoleCo-
            lumn);
        Database oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":name", DbType.String, user.
        name);
        oracle.AddInParameter(command, ":id", DbType.String, user.id);
        oracle.AddInParameter(command, ":role", DbType.String, user.role.
        id);
        return oracle.ExecuteNonQuery(command);
    }
}
```

（3）在数据访问层测试项目 DalTest 中，为 WebUserDB 添加一个单元测试类并运行。测试类代码如下：

```
[TestClass]
public class UserDBTest
{
    [TestMethod]
    public void testGetUserByID()                        //测试根据 ID 得到用户
    {
        string id = "admin";
        WebUser user = WebUserDB.getUserByID(id);
        Assert.IsNotNull(user);
        Assert.AreEqual(id, user.id);
    }
    [TestMethod]
    public void testUpdateUser()                             //测试修改用户
    {
        string id = "admin";
        WebUser user = WebUserDB.getUserByID(id);
        user.name = "管理员 0";
        int n = WebUserDB.updateUser(user);
        Assert.AreEqual(1, n);
        user = WebUserDB.getUserByID(id);
        Assert.AreEqual("admin", user.id);
        Assert.AreEqual("管理员 0", user.name);
    }
    [TestMethod]
    public void testAddUser()                                //测试添加用户
    {
        UserRole role = new UserRole() { id = "01" };
        WebUser user = new WebUser() { id = "new01", name = "new user", role
```

```
        = role };
        int n = WebUserDB.addUser(user);
        Assert.AreEqual(1, n);
        var user2 = WebUserDB.getUserByID("new01");
        Assert.IsNotNull(user2);
        Assert.IsTrue(user.password.Length > 0);
    }
    [TestMethod]
    public void testDeleteUser()                        //测试删除用户
    {
        string id = "new01";
        int n=WebUserDB.deleteUser(id);
        Assert.AreEqual(1, n);
        var user = WebUserDB.getUserByID(id);
        Assert.IsNull(user);
    }
}
```

（4）添加业务逻辑层类 WebUserBLL，代码与第 11 章相同，此处省略。

（5）在表现层项目 AccountCheckWeb 中添加一个页面 ManagerUser.aspx，页面代码与第 11 章相同，此处省略。

13.4.5　功能模块管理

功能模块管理可以对系统中的功能模块进行浏览、添加、删除、修改操作。实现功能模块管理的具体步骤如下。

（1）在业务实体项目 Account.Entity 中添加一个类 ApplicatonModule，以描述功能模块信息。

```
public class ApplicationModule
{
    public string id { get; set; }                      //模块编号
    public string name { get; set; }                    //模块名称
    public string description { get; set; }             //模块描述
    public string url { get; set; }                     //模块对应页面的 URL
}
```

（2）在数据访问层项目 AccountCheckDAL 中添加一个类 ApplicationModuleDB，以实现与功能模块相关的数据访问功能。

```
public static class ApplicationModuleDB
{
    //得到所有功能模块列表
    public static List<ApplicationModule> getAll()
    {
        string sql = "select ID, NAME, URL, DESCRIPTION from Applicati-
        on Module";
        var oracle = MyDbUtility.oracle;
        var reader = oracle.ExecuteReader(System.Data.CommandType.Text,
        sql);
        var list = new List<ApplicationModule>();
        while (reader.Read())
        {
            list.Add(fromDataReader(reader));
        }
        reader.Close();
```

```
        return list;
    }
    //根据模块编号得到模块对象
    public static ApplicationModule getByID(string id)
    {
        string sql = "select ID, NAME, URL, DESCRIPTION from Application
        Module where ID=:id";
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":id", DbType.String, id);
        var reader = oracle.ExecuteReader(command);
        ApplicationModule result = null;
        if (reader.Read())
        {
            result = fromDataReader(reader);
        }
        reader.Close();
        return result;
    }
    //根据页面 url 得到模块对象
    public static ApplicationModule getByUrl(string url)
    {
        string sql = "select ID, NAME, URL, DESCRIPTION from Application
        Module where url=:url";
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":url", DbType.String, url);
        var reader = oracle.ExecuteReader(command);
        ApplicationModule result = null;
        if (reader.Read())
        {
            result = fromDataReader(reader);
        }
        reader.Close();
        return result;
    }
    //添加模块数据
    public static int add(ApplicationModule module)
    {
        string sql = "insert into Application_Module (ID,NAME,DESCRIPTION,
        URL) values(:id,:name,:description,:url)";
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":id", DbType.String, module.id);
        oracle.AddInParameter(command, ":name", DbType.String, module.
        name);
        oracle.AddInParameter(command, ":description", DbType.String,
        module. description);
        oracle.AddInParameter(command, ":url", DbType.String, module.
        url);
        return oracle.ExecuteNonQuery(command);
    }
    //修改模块数据
    public static int update(ApplicationModule module)
    {
        string sql = "update Application Module set NAME=:name, DESCRIPTION=
        :description, URL=:url where ID=:id";
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":id", DbType.String, module.id);
        oracle.AddInParameter(command, ":name", DbType.String, module.
```

```
        name);
        oracle.AddInParameter(command, ":description", DbType.String,
        module. description);
        oracle.AddInParameter(command, ":url", DbType.String, module.
        url);
        return oracle.ExecuteNonQuery(command);
    }
    //根据模块 ID 删除模块
    public static int delete(string id)
    {
        string sql = "delete from Application Module where ID=:id";
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":id", DbType.String, id);
        return oracle.ExecuteNonQuery(command);
    }
    //从数据读取器的当前行数据生成一个功能模块对象
    private static ApplicationModule fromDataReader(IDataReader reader)
    {
        ApplicationModule module = new ApplicationModule();
        module.description = reader["description"].ToString();
        module.id = reader["id"].ToString();
        module.name = reader["name"].ToString();
        module.url = reader["url"].ToString();
        return module;
    }
}
```

（3）为 ApplicationModuleDB 类编写单元测试代码并运行测试，此处省略测试代码。

（4）编写业务逻辑层代码，此处省略具体代码。

（5）添加一个页面以显示数据和接受用户输入，此处省略页面的具体实现代码。

13.4.6　角色权限管理

社保卡结算系统使用的权限管理模块是一种基于角色的权限管理。一个角色拥有一组权限，用户属于哪个角色就拥有哪个角色的权限。系统管理员用户可以为其他角色分配权限。角色权限分配功能可以指定一个角色允许或者禁止访问某个模块，此功能具体实现步骤如下。

（1）在业务实体项目 Account.Entity 中添加一个类 ApplicationModule，以描述角色权限信息。

```
//角色权限业务实体类，描述了一个角色对一个功能模块的权限
public class RoleRight
{
    public string role { get; set; }                    //角色编号
    public string right { get; set; }       //1 表示拥有权限，0 表示没有权限
    public ApplicationModule module { get; set; }      //功能模块
}
```

（2）在数据访问层项目 AccountCheckDAL 中添加一个类 RoleRightDB，实现与角色权限相关的数据访问功能。

```
public static class RoleRightDB
{
```

```
#region public methods
/// <summary>
/// 得到某个角色的权限列表
/// </summary>
/// <param name="role">角色 ID</param>
/// <returns>权限列表</returns>
public static List<RoleRight> getRoleRight(string role)
{
    string sql = "select role id, module id, right from role right where
    role id=:role";
    var oracle = MyDbUtility.oracle;
    DbCommand command = oracle.GetSqlStringCommand(sql);
    oracle.AddInParameter(command, ":role", DbType.String, role);
    IDataReader reader = oracle.ExecuteReader(command);
    List<RoleRight> list = new List<RoleRight>();
    while(reader.Read())
    {
        list.Add(fromDataReader(reader));
    }
    reader.Close();
    return list;
}
/// <summary>
/// 判断某个角色是否可以访问某个页面
/// </summary>
/// <param name="role">角色 ID</param>
/// <param name="page">要访问的页面</param>
/// <returns>是否有权限访问</returns>
public static bool canAccessPage(string role, string page)
{
    var module = ApplicationModuleDB.getByUrl(page);
                                    //根据页面 URL 得到对应的模块
    if (module == null) return false;  //未打到指定模块
    string sql = "select right from md.role right where role id=:role
    and module id=:module";
    var oracle = MyDbUtility.oracle;
    DbCommand command = oracle.GetSqlStringCommand(sql);
    oracle.AddInParameter(command, ":role", DbType.String, role);
    oracle.AddInParameter(command, ":module", DbType.String, module.
    id);
    object o= oracle.ExecuteScalar(command);   //得到权限信息
    if (o == null || o == DBNull.Value) return true;
    return o.ToString() == "1";
}
/// <summary>
/// 修改角色权限
/// </summary>
/// <param name="roleRight">要修改的角色权限数据</param>
/// <returns>修改成功返回 1，否则 0</returns>
public static int update(RoleRight roleRight)
{
    string sql = "update md.role right set right=:right where module id=
    :module and role id=:role";
    var oracle = MyDbUtility.oracle;
    DbCommand command = oracle.GetSqlStringCommand(sql);
    oracle.AddInParameter(command, ":role", DbType.String, roleRight.
```

```
        role);
        oracle.AddInParameter(command, ":right", DbType.String, roleRi-
        ght.right);
        oracle.AddInParameter(command, ":module", DbType.String,
        roleRight. module.id);
        return oracle.ExecuteNonQuery(command);
    }
    /// <summary>
    /// 刷新角色权限，删除过期数据，为新角色和新模块生成默认数据
    /// </summary>
    /// <returns>新产生的角色权限数据记录数</returns>
    public static int refreshRoleRight()
    {
        string sql;
        //删除角色权限表中不存在的角色数据
        sql = "delete from md.role right where role id not in(select id from
        md.user role)";
        MyDbUtility.oracle.ExecuteNonQuery(System.Data.CommandType.Text,
        sql);
        //删除角色权限表中不存在的模块数据
        sql = "delete from md.role_right where module_id not in(select id
        from md.application_module)";
        MyDbUtility.oracle.ExecuteNonQuery(System.Data.CommandType.Text,
        sql);
        sql = "update md.role_right set right='0' where right<>'0' and
        right<>'1'";
        MyDbUtility.oracle.ExecuteNonQuery(System.Data.CommandType.Text,
        sql);
        //为新的角色和新的模块生成角色权限数据，并赋默认值为 0（没有权限）
        sql=@"insert into role_right(role_id,module_id,right)
select r.id as rid,m.id as mid,'0' as right2
from user role r  cross join  application module m
where not exists (select * from role_right rr where rr.role_id=r.id and
rr.module id=m.id )";
        return MyDbUtility.oracle.ExecuteNonQuery(System.Data.Command-
        Type. Text, sql);
    }
    #endregion
    #region private methods
    /// <summary>
    /// 从数据读取器当前记录产生一个角色权限对象
    /// </summary>
    /// <param name="reader">数据读取器</param>
    /// <returns>所产生的角色权限对象</returns>
    private static RoleRight fromDataReader(IDataReader reader)
    {
        RoleRight result = new RoleRight();
        result.role = reader["role id"].ToString();
        result.right = reader["right"].ToString();
        string temp = reader["module id"].ToString();
        result.module = ApplicationModuleDB.getByID(temp);
        return result;
    }
    #endregion
}
```

（3）为 RoleRightDB 类编写单元测试代码并运行测试，此处省略测试代码。
（4）编写与角色权限相关的业务逻辑层代码，此处省略具体代码。
（5）添加一个页面以显示数据和接收用户输入，此处省略页面的具体实现代码。

13.4.7　医疗机构权限管理

根据用户需求，社保卡结算系统的用户只可以访问被允许的医疗机构的数据。医疗机构用户只可以访问本机构的数据，而社保机构用户则需要系统管理员为其分配可访问的医疗机构列表。这种权限管理方式称为医疗机构权限管理。根据用户需求，即使同一个角色的不同用户所允许访问的医疗机构列表也不相同，因此医疗机构权限管理是基于用户而非角色进行控制的。下面将介绍医疗机构权限管理的具体实现步骤。

（1）在业务实体项目 Account.Entity 中添加一个类 HospitalRight，以描述用户医院权限。

```
public class HospitalRight
{
    public string userId { get; set; }              //用户 ID
    public string hospitalNo { get; set; }          //医疗机构编码
    public string hostpitalName { get; set; }       //医疗机构名称
    public string InstituteNo { get; set; }         //社保机构编号
    public string right { get; set; }               //权限
    //根据医疗机构实体对象设计各个字段的值
    public void setHospital(Hospital hospital)
    {
        this.hostpitalName = hospital.Name;
        this.InstituteNo = hospital.InstituteNo;
        this.hospitalNo = hospital.HospitalNo;
    }
}
```

（2）在数据访问层项目 AccountCheckDB 中添加一个类 HospitalRightDB，实现与医疗机构权限相关的数据访问功能。

```
public static class HospitalRightDB
{
    /// <summary>
    /// 刷新用户医院权限数据，为新医疗机构和新用户插入数据
    /// </summary>
    /// <returns>新添加的数据行数</returns>
    public static int refreshHospitalRight()
    {
        string sql;
        //将所有权限既不为 0 也不为 1 的非法数据全部设置为 0
        sql = "update md.user hospital right set right='0' where right<>'1'
        and right<>'0'";
        MyDbUtility.oracle.ExecuteNonQuery(System.Data.CommandType.Text,
        sql);
        //添加新医医疗机构和新用户的权限（默认为 0）
        sql = @"insert into md.user hospital right
                (user id,yybm,sbjgbh,right)
                select r.id as user id2,m.yybm,m.sbjgbh,'0' as right2
                from md.web_user r  cross join  md.institution_natl m
                where not exists
                (select * from md.user_hospital_right wu
```

```
                where wu.user_id=r.id and wu.yybm=m.yybm and wu.sbjgbh=
                m.sbjgbh )";
        return MyDbUtility.oracle.ExecuteNonQuery(System.Data.Command-
        Type. Text, sql);
    }
    /// <summary>
    /// 根据用户名和搜索关键字查找医院权限
    /// </summary>
    /// <param name="user">用户 ID</param>
    /// <param name="key">搜索关键字（医院名称或者医院编码）</param>
    /// <param name="byName">根据名称还是编码，true 名称，false 编码</param>
    /// <param name="page">分页参数</param>
    /// <returns>查找到的医院权限列表</returns>
    /// <remarks>
    /// 如果搜索关键字(参数 key)不为空，则查找指定用户对符合条件的医疗机构的权限，
    /// 如果搜索关键字为空，则查找指定用户对所有医疗机构的权限
    /// </remarks>
    public static List<HospitalRight> getUserHospitalRight(string user,
    string key, bool byName, PageDataArgument page)
    {
        string sql = null;
        //如果没有搜索条件则直接查询
        if (string.IsNullOrEmpty(key))
        {
            sql = @"select * from (
                  select t1.yybm,t1.sbjgbh,t1.right,t2.yymc,rownum as rn
                  from md.user_hospital_right t1 inner join md.INSTITUTION_
                  NATL t2
                  on t1.yybm=t2.yybm and t1.sbjgbh=t2.sbjgbh
                  where t1.user id=:u0 order by t1.sbjgbh,t1.yybm)";
        }
        else
        {
            //根据医院名称查找
            if (byName)
            {
                sql = @"select * from (
                      select t1.yybm,t1.sbjgbh,t1.right,t2.yymc,rownum as-
                      rn from md.user hospital right t1 inner join md.INSTI-
                      TUTION_NATL t2
                      on t1.yybm=t2.yybm and t1.sbjgbh=t2.sbjgbh
                      where t1.user id=:u0 and t2.yymc like :key order by
                      t1.sbjgbh,t1.yybm)";
            }
            //根据医院编码查找
            else
            {
                sql = @"select * from (
                      select t1.yybm,t1.sbjgbh,t1.right,t2.yymc,rownum as-
                      rn from md.user hospital right t1 inner join md.
                      INSTITUTION NATL t2
                      on t1.yybm=t2.yybm and t1.sbjgbh=t2.sbjgbh
                      where t1.user_id=:u0 and t1.yybm like :key order by
                      t1.sbjgbh,t1.yybm)";
            }
        }
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":u0", DbType.String, user);
```

```
        //有搜索条件则添加到 where 子句中
        if (!string.IsNullOrEmpty(key))
            oracle.AddInParameter(command, ":key", DbType.String, "%" + key
            + "%");
        if (page.refreshCount)
            page.count = MyDbUtility.getCount(command);
        command.CommandText = sql + " where rn  between "
            + page.min + " and " + page.max; //生成分页条件
        IDataReader reader = oracle.ExecuteReader(command);
        List<HospitalRight> result = new List<HospitalRight>();
        //循环读取数据并添加到列表中
        while (reader.Read())
        {
            HospitalRight r = new HospitalRight();
            r.hospitalNo = reader[0].ToString();
            r.InstituteNo = reader[1].ToString();
            r.hostpitalName = reader["yymc"].ToString();
            r.right = reader["right"].ToString();
            r.userId = user;
            result.Add(r);
        }
        return result;
    }
    //得到指定用户的医疗机构权限列表
    public static List<HospitalRight> getUserHospitalRight(string user,
    PageDataArgument page)
    {
        return getUserHospitalRight(user, null,false,page);
    }
    public static List<Hospital> getGrantedHospitals(string user,string
    yybm, PageDataArgument page)
    {
        return getHospitalsByRight(user, yybm, true, page);
    }
    //得到指定用户被授权的医疗机构列表
    public static List<Hospital> getGrantedHospitals(string user,Page-
    DataArgument page)
    {
        return getHospitalsByRight(user,null,true,page);
    }
    //得到指定用户未被授权的医疗机构列表
    public static List<Hospital> getUngrantedHospitals(string user,Page-
    DataArgument page)
    {
        return getHospitalsByRight(user,null, false,page);
    }
    /// <summary>
    /// 根据权限分类得到指定用户的医疗机构列表
    /// </summary>
    /// <param name="user">指定用户</param>
    /// <param name="yybm">医疗机构编码</param>
    /// <param name="can">是否可以访问</param>
    /// <param name="page">分页参数</param>
    /// <returns>得到的医疗机构列表</returns>
    private static List<Hospital> getHospitalsByRight(string user,string
    yybm, bool can,PageDataArgument page)
    {
        string r = can ? "1" : "0";
        string sql =null;
```

```
        //是否有过滤条件（根据医院编码过滤）
        bool filter=!string.IsNullOrEmpty(yybm);/
        if (filter)
            sql = " select * from (select yybm,sbjgbh,rownum as rn from
                   md.user hospital right"+" where right=:r0 and user id=:u0
                   and yybm like :yybm) ";
        else
            sql = "select * from (select yybm,sbjgbh,rownum as rn from
                   md.user_hospital_right "+ " where right=:r0 and user_id=:
                   u0)";
        var oracle = MyDbUtility.oracle;
        //创建命令并添加参数
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":r0", DbType.String, r);
        oracle.AddInParameter(command, ":u0", DbType.String, user);
        if (filter)
            oracle.AddInParameter(command, ":yybm", DbType.String, "%"
            +yybm+"%");
        //得到记录总数
        if (page.refreshCount)
            page.count = MyDbUtility.getCount(command);
        //设置分页查询命令
        command.CommandText = sql + " where rn  between " + page.min + " and
        " + page.max;
        //读取数据并构建实体对象列表
        IDataReader reader = oracle.ExecuteReader(command);
        List<Hospital> result = new List<Hospital>();
        while (reader.Read())
        {
            Hospital item  = HospitalDB.getByID(reader[0].ToString(),
            reader[1].ToString());
            result.Add(item);
        }
        return result;
    }
    //修改医院权限数据
    public static int update(HospitalRight right)
    {
        //创建 update 语句
        string sql = "update md.user_hospital_right set right=:r0 where
        user_id=:u0 and sbjgbh=:sbjgbh and yybm=:yybm";
        var oracle = MyDbUtility.oracle;
        //创建命令，添加参数，执行命令
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":u0", DbType.String, right.user-
        Id);
        oracle.AddInParameter(command, ":sbjgbh", DbType.String, right.
        InstituteNo);
        oracle.AddInParameter(command, ":yybm", DbType.String,right.
        hospitalNo);
        oracle.AddInParameter(command, ":r0", DbType.String, right.
        right);
        return oracle.ExecuteNonQuery(command);
    }
}
```

（3）编写单元测试以测试 AccountCheckDB.HospitalRightDB 类。此处省略测试代码。

（4）在业务逻辑层项目 AccountCheckBLL 中添加一个类 HospitalRightBLL，这个类的主要代码是调用数据访问层 AccountCheckDB.HospitalRightDB 类的相应方法。此处省略

HospitalRightBLL 类的代码。

（5）在表现层项目 AccountCheckWeb 中，添加一个页面 HospitalRightPage.aspx，此页面可以查看和修改用户对医院的权限，页面设计外观如图 13.14 所示。



图 13.14　医院权限页面设计外观

图 13.14 所示 HospitalRightPage.aspx 页面布局较为复杂，下面对页面功能和使用做一个简单说明。从用户列表中选择一个用户然后单击“查看”按钮，则在页面下方的 GridView 中显示此用户对所有医疗机构的权限。由于医疗机构众多，从 GridView 列表中直接查找某个医疗机构很不方便，此时可以通过输入医院编码或者医院名称的方式进行过滤，快速找到某个医院。可以在 GridView 中一次选中多个医疗机构，对其进行批量授权或者取消授权。HospitalRightPage.aspx 页面代码如下：

```
<%@ Register assembly="AspNetPager" namespace="Wuqi.Webdiyer" tagprefix
="webdiyer" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<%--导入 JavaScript 文件--%>
<script src="../js/jquery-1.3.2.js" type="text/javascript"></script>
<script src="../js/jquery-ui-1.7.2.js" type="text/javascript"></script>
    <script src="../js/MyUtility.js" type="text/javascript"></script>
    <script type="text/javascript">
        $(function () {
            //为 GridView 添加光棒效果
            $('table.gridview').find("tr").hover(
            function() { $(this).addClass('hoverRow'); },
            function() { $(this).removeClass('hoverRow'); }
            ); //$('table').tr.hover
            $SjlUtility.addButtonClass();
            //为全选 CheckBox 添加事件处理程序
            $('#selectall').change(checkAllChanged);
        }); //$(function)
        //当全选 CheckBox 状态发生改变时，更改所有数据行的 CheckBox 状态
        function checkAllChanged()
        {
            var b = this.checked;
            $("input:checkbox", $('table.gridview'))
            .each(function()
```

```
            { this.checked = b; }
            );
        }
      </script>
</asp:Content>
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
   <!-- 用户列表 -->
   用户：<asp:DropDownList ID="userList" runat="server"></asp:DropDown-
   List>
<asp:Button ID="ok" runat="server" Text="查看" onclick="ok_Click" />  
每页记录数<asp:TextBox ID="pageSize" runat="server" Width="40px">10</asp:
TextBox>
<asp:Button ID="Button1" runat="server" Text="修改" onclick="Button1_
Click" />  <br />
 筛选条件：
 医院编码<asp:TextBox ID="hospitalCode" runat="server"></asp:TextBox>
 医院名称<asp:TextBox ID="hospitalName" runat="server"></asp:TextBox>
<asp:Button ID="filter" runat="server" Text="筛选" onclick="filter_Click"
/><br />
对选中的医院批量授权
<asp:DropDownList ID="rightList" runat="server">
   <asp:ListItem Selected="True" Value="1">允许</asp:ListItem>
   <asp:ListItem Value="0">拒绝</asp:ListItem>
</asp:DropDownList> 
<asp:Button ID="Button2" runat="server" Text="确定" onclick="Button2_
Click" />
<br /> <br />
<asp:HiddenField ID="hiddenUser" runat="server" />
<asp:GridView ID="GridView1" runat="server" CssClass="gridview"
   AutoGenerateColumns="False"   DataKeyNames="InstituteNo,Hospital-
   No" >
   <Columns>
      <asp:BoundField  DataField="InstituteNo" HeaderText="社保机构编号"
      ItemStyle-Width="100" />
    <asp:BoundField DataField="HospitalNo" HeaderText="医院编码" Item-
    Style-Width="100" />
      <asp:BoundField DataField="HospitalName" HeaderText="医院名称" Item-
      Style-Width="300" />
      <asp:TemplateField HeaderText="权限" ItemStyle-Width="60">
      <ItemTemplate>
         <asp:Label ID="theRight" runat="server" Text='<%#Eval("right")
         .ToString()=="1"?"有":"无" %>'></asp:Label>
      </ItemTemplate>
      </asp:TemplateField>
      <asp:BoundField DataField="right" Visible="false" />
      <asp:TemplateField HeaderText="操作" Visible="false" >
         <ItemTemplate>
         <asp:LinkButton runat="server" ID="operation" Text="允许_禁止"
         CommandName="grant deny" />
          </ItemTemplate>
         </asp:TemplateField>
         <asp:TemplateField >
         <HeaderTemplate>
         <!-- 在 GridView 表格头部显示一个全选 CheckBox-->
         <input type="checkbox" id="selectall" />
         </HeaderTemplate>
         <ItemTemplate>
```

```
                <asp:CheckBox ID="selectCheckBox" runat="server" />
            </ItemTemplate>
            </asp:TemplateField>
    </Columns>
    </asp:GridView>
    <%-- 分页控件 --%>
    <webdiyer:AspNetPager ID="pager1" runat="server"
        onpagechanged= "pager1_PageChanged"> </webdiyer:AspNetPager>
</asp:Content>
```

（6）在 HospitalRightPage.aspx 页面的 Page_Load 事件中，初始化数据绑定，包括刷新医院权限数据和绑定用户列表。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        HospitalRightBLL.refreshHospitalRight();   //刷新医院权限数据
        bindList();
    }
}
//绑定用户列表
private void bindList()
{
    List<WebUser> list= WebUserBLL.getAllUsers();  //得到所有用户
    foreach (WebUser item in list)
    {
        ListItem li = new ListItem();
        li.Text = item.name + "[" + item.id+"]";
        li.Value = item.id;
        userList.Items.Add(li);                    //将用户添加到下拉列表中
    }
    userList.SelectedIndex = 0;
}
```

（7）在 HospitalRightPage.aspx 页面的“查看”按钮的 Click 事件中，查找并显示指定用户对所有医院的权限数据。

```
protected void ok_Click(object sender, EventArgs e)
{
    hospitalName.Text = "";
    hospitalCode.Text = "";
    pager1.CurrentPageIndex = 1;
    bindRights(true);
}
/// <summary>
/// 绑定医院权限列表到 GridView 控件
/// </summary>
/// <param name="refreshCount">是否刷新总记录数</param>
private void bindRights(bool refreshCount)
{
    string user = userList.SelectedValue;          //得到下拉列表选中的用户
    hiddenUser.Value = user;
    PageDataArgument arg = new PageDataArgument();
    arg.refreshCount = refreshCount;
    arg.pageIndex = pager1.CurrentPageIndex - 1;
    arg.pageSize = pager1.PageSize;
    List<HospitalRight> list = null;
    bool filter = hospitalCode.Text.Length > 0
```

```
        || hospitalName.Text.Length > 0;            //是否有过滤条件
    if (filter)
    {
        if (hospitalCode.Text.Length > 0)           //根据医院编码过滤
            list=HospitalRightBLL.getUserHospitalRight(user, hospital-
            Code. Text.Trim(), false, arg);
        else
            if (hospitalName.Text.Length > 0)       //根据医院名称过滤
                list=HospitalRightBLL.getUserHospitalRight(user, hospit-
                alName.Text.Trim(), true, arg);
    }
    else
         list=HospitalRightBLL.getUserHospitalRight(user, arg);
                                                    //没有过滤条件
    if(refreshCount)
        pager1.RecordCount = arg.count;
    GridView1.DataSource = list;
    GridView1.DataBind();
}
```

（8）在分页控件的 PageChanged 事件中，重新绑定新的一页数据。

```
protected void pager1 PageChanged(object sender, EventArgs e)
{
    bindRights(false);
}
```

（9）在“确定”按钮的 Click 事件中，批量修改选择医院的权限。

```
protected void Button2 Click(object sender, EventArgs e)
{
    bool changed = false;
    string right = rightList.SelectedValue;     //有无权限
    string user = hiddenUser.Value;             //要修改权限的用户
    for (int i = 0; i < GridView1.Rows.Count; i++)
                                                //针对 GridView 中所有行循环
    {
        CheckBox c = GridView1.Rows[i].FindControl("selectCheckBox") as
        CheckBox;
        if (!c.Checked) continue;               //未选中该行则继续下一行
        HospitalRight item = new HospitalRight();
        item.right = right;
        item.InstituteNo = GridView1.DataKeys[i]["InstituteNo"].ToStr-
        ing();
        item.hospitalNo = GridView1.DataKeys[i]["HospitalNo"].ToString();
        item.userId = user;
        HospitalRightBLL.update(item);          //修改授权
        changed = true;
    }
    if (changed)
        bindRights(true);
}
```

（10）在“筛选”按钮的 Click 事件中，根据指定的医院编码或医院名称筛选并显示医院权限数据。

```
protected void filter Click(object sender, EventArgs e)
{
    bindRights(true);
}
```

（11）在浏览器中查看 HospitalRightPage.aspx 页面，显示界面如图 13.15 所示。

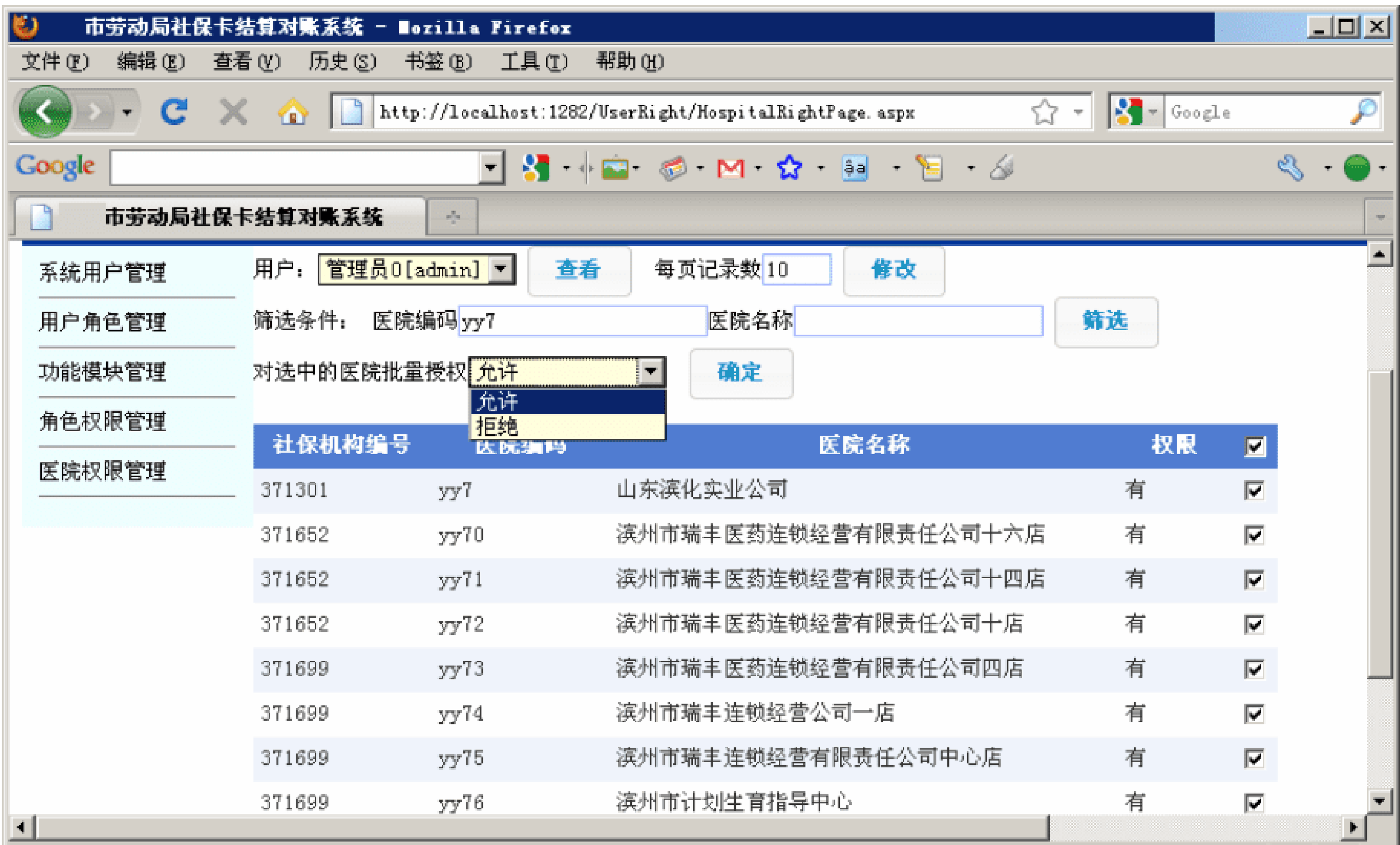


图 13.15　医院权限页面

13.4.8　用户登录

社保卡结算系统仅对 A 市人力资源与社会保障局及定点医疗机构人员开放，使用该系统必须具有合法的用户名和密码，先登录再使用。用户登录成功后，需要把登录信息保存在 Session 中，并在其他页面中根据此信息进行用户权限检测和其他操作。在表现层项目 AccountCheckWeb 中添加一个 Login.aspx 页面。页面布局如图 13.16 所示。



图 13.16　登录页面

在登录页面 Login.aspx 中，为了获得较好的用户体验，用 JavaScript 添加了一些特效，如果鼠标移动到按钮上会变色，鼠标移动到文本框时文本框会自动获得焦点并选中其中文本。Login.aspx 页面代码如下：

```
<head runat="server">
   <title>A 市劳动局医保数据对账系统</title>
   <link type="text/css" rel="Stylesheet" href="css/login.css" />
   <script src="js/jquery-1.3.2.js" type="text/javascript"></script>
    <script type="text/javascript">
       $(function() {
          checkIE6();
          checkResolution();
             //当鼠标移动到用户名和密码文本框时，自动获得焦点，并选中其中文本
             $("input#userName").mouseover(focusSelect);
             $("input#password").mouseover(focusSelect);
             //为登录按钮添加样式
             $('input#login2').addClass("ui-button ui-state-default ui-
             corner-all");
             $('input#login2').hover(function ()
             { $(this).addClass("ui-state-hover"); },
             function() { $(this).removeClass("ui-state-hover"); });
          }
       );
       //检查浏览器版本（不支持 IE6 及以下）
       function checkIE6() {
        if ($.browser.msie) {
            if($.browser.version <= 6)
            alert("你使用的浏览器太旧（IE6 及以下）了，请使用新的浏览器（如 IE
            8,FireFox,Chrome 等");
        }
        }
       //检查屏幕分辨率
       function checkResolution()
       {
         if(screen.width<1000)
            alert('请将屏幕分辨改为 1024 以上。');
      }
      //使控件获得焦点，并选中控件中的所有文本
      function focusSelect() {
         this.focus();
         this.select();
      }
    </script>
</head>
<body>
   <form id="form1" runat="server">
   <div id="main">
   <div>
     <img src="images/logo.jpg" alt="" />
     </div>
 <div id="login">
       用户 ID <asp:TextBox ID="userName" runat="server" AutoCompleteType=
       "None" > </asp:TextBox>
    密 码 <asp:TextBox ID="password" runat="server" TextMode="Pass-
    word"> </asp:TextBox>
    <asp:Button  ID="login2"  runat="server"  Text=" 登 录 "  Width="80"
    Height="25"  onclick="login2_Click"/>
    </div>
```

```
    </div>
    </form>
</body>
```

在“登录”按钮的 Click 事件中，验证用户输入的用户名和密码，如果正确，则跳转到默认页面。由于社保卡结算系统有两类用户：社会保障局工作人员和医疗机构工作人员，这两类用户登录使用的用户名和密码也不相同，所以在验证用户登录时要注意这一点，代码如下：

```
protected void login2_Click(object sender, EventArgs e)
{
    bool success = false;
    WebUser user=WebUserBLL.getByID(userName.Text);
                                        //根据输入的用户名查找用户信息
    if (user == null)                   //未找到用户
    {
        //判断是否医疗机构
        Hospital h=HospitalBLL.getByID(userName.Text, password.Text);
        if(h!=null)                     //是医疗机构
        {
            success=true;
            user=new WebUser(){ id=userName.Text, name=h.Name};
            user.role = UserRole.HospitalRole;
            WebUtility.instituteId=password.Text;
        }
    }
    else                                //找到了用户
    {
        if (user.password == password.Text)//密码是否正确
        {
            success = true;
            WebUtility.instituteId = "";
        }
    }
    if(success)                         //登录成功则跳转到默认页面
     {
         Common.WebUtility.currentUser = user;
         Response.Redirect("default/CheckAccount2.aspx");
    }
    else                                //登录失败则提示错误信息
    {
        ClientScript.RegisterStartupScript(GetType(), "errorlogin",
            "<script>alert('用户名或密码错误！');</script>");
    }
}
}
```

13.5　银行数据上传

在社保卡结算系统中，社保卡的消费数据由银行提供。银行定期向 A 市人力资源与社会保障局提供一个该时间段内社保卡消费明细（一个 dbf 文件），这个文件通过社保卡结算系统导入到 A 市人力资源与社会保障局的 Oracle 数据库中，并以此数据为基础实现结算和

报表功能。在数据上传时，除上传银行数据本身以外，还要将上传日志保存起来，以备后续查询。

13.5.1　数据访问层和业务逻辑层

数据访问层和业务逻辑层的功能是将已经上传到服务器的指定 dbf 文件中的数据导入到 Oracle 数据库，并记录上传日志。这个操作过程需要用到 Oracle 数据库中的两个表：银行卡消费明细表 MD.BANK_TRANSACTION 和数据上传日志表 MD.UPLOAD_LOG，在数据访问层编写两个类分别操作这两个表。

（1）在数据访问层项目 AccountCheckDB 中，添加一个类 DbfTransfer，用于将 dbf 文件中的数据导入 MD.BANK_TRANSACTION 表。代码如下：

```
//将数据从 dbf 文件中导入到 Oracle 数据库的 md.Bank_transacton 表
public static class DbfTransfer
{

    /// <summary>
    /// 将数据从 dbf 文件中导入到 Oracle 数据库的 md.Bank_transacton 表
    /// </summary>
    /// <param name="file">数据来源 dbf 文件（包含完整路径和文件名）</param>
    /// <param name="success">成功导入的数据行数（输出参数）</param>
    /// <param name="failure">导入失败的数据行数（输出参数）</param>
    /// <returns>是否全部导入成功</returns>
    public static bool transferData(string file,out int success,out int
    failure)
    {
        success = 0;
        failure = 0;
        //创建一个 Odbc 连接对象用于打开 dbf 文件
        System.Data.Odbc.OdbcConnection oConn = new System.Data.Odbc.
        OdbcConnection();
        string s = "Driver={Microsoft dBase Driver (*.dbf)};SourceType=DBF;
        SourceDB=" + file+ "; Exclusive=No; Collate=Machine; NULL=NO;
            DELETED=NO; BACKGROUNDFETCH=NO;";
        oConn.ConnectionString = s;
        System.Data.Odbc.OdbcCommand oCmd = oConn.CreateCommand();
        oCmd.CommandText = "SELECT * FROM "+file;
        Database oracle = MyDbUtility.oracle;
        //生成插入数据的 SQL 语句
        DbCommand command =
            oracle.GetSqlStringCommand("insert into md.Bank Transaction "
            + "(ID, WORKDATE, CORNO, NAME, ACCNO, AMT, FEE, REALAMT, TYPE,
            TERM, TIME, CSERNO) VALUES "
        +"(:id,:workdate,:corno,:name,:accno,:amt,:fee,:realamt,:type,
        :term,:time,:cserno)");
        //向命令添加各个参数
        oracle.AddInParameter(command, ":id", DbType.String);
        oracle.AddInParameter(command, ":workdate", DbType.String);
        oracle.AddInParameter(command, ":corno", DbType.String);
        oracle.AddInParameter(command, ":name", DbType.String);
        oracle.AddInParameter(command, ":accno", DbType.String);
        oracle.AddInParameter(command, ":amt", DbType.Currency);
        oracle.AddInParameter(command, ":fee", DbType.Currency);
        oracle.AddInParameter(command, ":realamt", DbType.Currency);
```

```
        oracle.AddInParameter(command, ":type", DbType.String);
        oracle.AddInParameter(command, ":term", DbType.String);
        oracle.AddInParameter(command, ":time", DbType.DateTime);
        oracle.AddInParameter(command, ":cserno", DbType.String);
        DateTime now=DateTime.Now;
        int year= now.Year;
        int n = 0;
        //生成上传数据 ID 的前缀（根据当前日期时间生成）
        string idPrefix = string.Format("{0:0000}", now.Year)
            + string.Format("{0:00}", now.Month)
            + string.Format("{0:00}", now.Day) + string.Format("{0:00}",
            now.Hour)
            + string.Format("{0:00}", now.Minute) + string.Format("{0:
            00}",now.Second);
        int idSuffix = 0;                           //上传数据 ID 的后缀
        //上传的 dbf 数据表中所有列
        string[] columns = {"workdate", "corno","name","accno","amt",
        "fee","realamt","type", "term", "cserno" };
        try
        {
            oConn.Open();
            //从 dbf 文件中循环读取每行数据，并插入到 Oracle 中
            using (IDataReader reader = oCmd.ExecuteReader (CommandBehav-
            ior.CloseConnection ))
            {
                while (reader.Read())
                {
                    string timeStr = "";
                    int month = 0, day = 0, hour = 0, minute = 0, second =
                    0;
                    try
                    {
                        for (int i = 0; i < columns.Length; i++)
                        {
                            oracle.SetParameterValue(command, ":"+columns
                            [i], reader[columns[i]]);
                        }
                        //当后缀达到最大值时，后缀清零，并重新生成前缀
                        if (idSuffix >= 999998)
                        {
                            idSuffix = 0;
                            idPrefix = string.Format("{0:0000}", now.Year)
                            + string.Format("{0:00}", now.Month) + string.
                            Format("{0:00}", now.Day)
                                + string.Format("{0:00}", now.Hour) +
                                string.Format("{0:00}", now.Minute) +
                                string. Format("{0:00}", now.Second);
                        }
                        oracle.SetParameterValue(command, ":id", idPrefix
                        +string.Format("{0:000000}",(idSuffix++).ToStr-
                        ing()));
                        //银行提供的 dbf 文件中时间为字符型，格式为 mmddhhmiss
                        //需要转换为日期型
                        timeStr = reader["time"].ToString();
                        month = int.Parse(timeStr.Substring(0, 2));
                        day = int.Parse(timeStr.Substring(2, 2));
                        hour = int.Parse(timeStr.Substring(4, 2));
                        minute = int.Parse(timeStr.Substring(6, 2));
                        second = int.Parse(timeStr.Substring(8, 2));
                    }
```

```
                    //如果发生异常，传输失败计数加 1
                    catch
                    {
                        failure++;
                        continue;
                    }
                    try
                    {
                        oracle.SetParameterValue(command, ":time", new
                        DateTime(year, month, day, hour, minute, second));
                        n = oracle.ExecuteNonQuery(command);
                        success++;
                    }
                    //如果发生异常，传输失败计数加 1
                    catch (OracleException ex)
                    {
                        failure++;
                    }
                    catch (Exception ex)
                    {
                        failure++;
                    }
                }
            }
        }
        finally
        {
            oConn.Close();
            command.Dispose();
        }
        return failure==0;
    }
}
```

（2）在测试项目 DalTest 中为 DbfTransfer 类创建单元测试并执行，以检查和改正 DbfTransfer 中的错误。

（3）在数据访问层项目 AccountCheckDAL 中添加一个类 UploadLogDB，以添加和查询数据上传日志。代码如下：

```
public static class UploadLogDB
{
    /// <summary>
    /// 添加一个数据上传日志
    /// </summary>
    /// <param name="log">要添加的日志</param>
    /// <returns>添加的日志数量</returns>
    public static int add(UploadLog log)
    {
        //构建 insert 语句
        string sql = "insert into MD.UPLOAD LOG "
            +"(User id, total record, correct record, error record)"
            +" values (:userid, :total, :correct, :err)";
        var oracle = MyDbUtility.oracle;
        //创建一个命令，并添加各个参数
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":userid",DbType.String,log.
        userId);
        oracle.AddInParameter(command, ":total", DbType.Int32, log.total-
        Record);
```

```
        oracle.AddInParameter(command, ":correct", DbType.Int32, log.
        correctRecord);
        oracle.AddInParameter(command, ":err", DbType.Int32, log. Error-
        Record);
        return oracle.ExecuteNonQuery(command);            //执行命令
    }
    /// <summary>
    /// 根据时间范围查询上传日志情况
    /// </summary>
    /// <param name="date">日期范围</param>
    /// <param name="page">分页参数</param>
    /// <returns>查询到的符合条件的日志列表</returns>
    public static List<UploadLog> getByDate(DateRangeArg date, PageData-
    Argument page)
    {
        string sql = "select upload id, user id, upload date,"
            +" total record,correct record,error record, rownum as rn "
            +" from MD.UPLOAD LOG "
            +" where Upload_Date between :date1 and :date2";
        var oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":date1", DbType.DateTime, date.
        from);
        oracle.AddInParameter(command, ":date2", DbType.DateTime, date.
        to);
        //如果需要，则刷新数据总行数
        if (page.refreshCount)
            page.count = MyDbUtility.getCount(command);
        //构建分页查询语句
        command.CommandText = "select * from (" +
            sql + ") where rn >"+page.pageIndex*page.pageSize
            + " and rn<=" + (page.pageIndex+1) * page.pageSize;
        IDataReader reader = oracle.ExecuteReader(command);
        //循环读取数据，创建对象列表
        List<UploadLog> list = new List<UploadLog>();
        while (reader.Read())
        {
            UploadLog log = new UploadLog();
            log.id = Convert.ToInt32(reader["upload id"]);
            log.userId = Convert.ToString(reader["user id"]);
            log.date = Convert.ToDateTime(reader["upload date"]);
            log.totalRecord = Convert.ToInt32(reader["total_record"]);
            log.correctRecord = Convert.ToInt32(reader ["correct_
            record"]);
            log.errorRecord = Convert.ToInt32(reader["error record"]);
            list.Add(log);
        }
        reader.Close();
        return list;
    }
}
```

（4）在测试项目 DalTest 中为 UploadLogDB 类创建单元测试并执行，以检查和改正 DbfTransfer 中的错误。测试代码如下：

```
[TestClass()]
public class UploadLogDBTest
{
```

```
    [TestMethod()]
    public void addTest()                               //测试添加日志
    {
        UploadLog log = new UploadLog()
        {   userId = "admin", totalRecord = 5000,
            correctRecord = 4900, errorRecord = 100 };
                                                        //创建一个上传日志
        int actual = UploadLogDB.add(log);
        Assert.AreEqual(1, actual);                     //方法应该返回 1
    }
    [TestMethod]
    public void getByDateTest()
    {
        PageDataArgument page = PageDataArgument.all;
        page.refreshCount = true;                       //得到所有数据并获取总数
        //设定日期范围
        DateRangeArg date = DateRangeArg.between2Date("2010-4-1", "2010-
        4-30");
        var list = UploadLogDB.getByDate(date,page);
        Assert.IsTrue(list.Count > 0);
        Assert.IsTrue(list.Count == page.count);
        Assert.IsNotNull(list[0]);                      //列表中的数据不为空
    }
}
```

13.5.2　数据上传页面

在数据上传页面中，用户可以选择一个 dbf 文件，将文件上传到服务器，然后把数据从 dbf 文件中转入到 Oracle 数据库中。由于银行提供的数据文件较大，上传和导入需要一定时间，在实现数据上传和导入功能时，需要充分考虑这两个问题。实现数据上传和导入页面的具体步骤如下。

（1）ASP.NET Web 应用程序对上传文件大小有限制，默认情况下只允许上传小于 2M 的文件。但银行提供的社保卡消费数据 dbf 文件大都超过 2M，需要修改 web.config 配置文件中 system.web 结点的两个选项以允许上传大文件。

```
<system.web>
<!--maxRequestLenght 限制上传文件大小，以 KB 为单位-->
<!--允许执行请求的最大时间限制，单位为秒-->
<httpRuntime maxRequestLength="10240" executionTimeout="120"/>
…
</system.web>
```

（2）添加一个页面 UploadData.aspx，该页面主要包含一个文件上传控件、一个按钮和一个显示上传进度的动画图片。页面布局如图 13.17 所示。

在 UploadData.aspx 页面中，默认情况下动画图片不显示，只有当数据正在上传和导入时动画才显示。在页面上用一个隐藏字段 HiddenField 来标识是否有数据正在上传，并以此标志来判断是否显示动画图片。UploadData.aspx 页面代码如下：

```
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
```

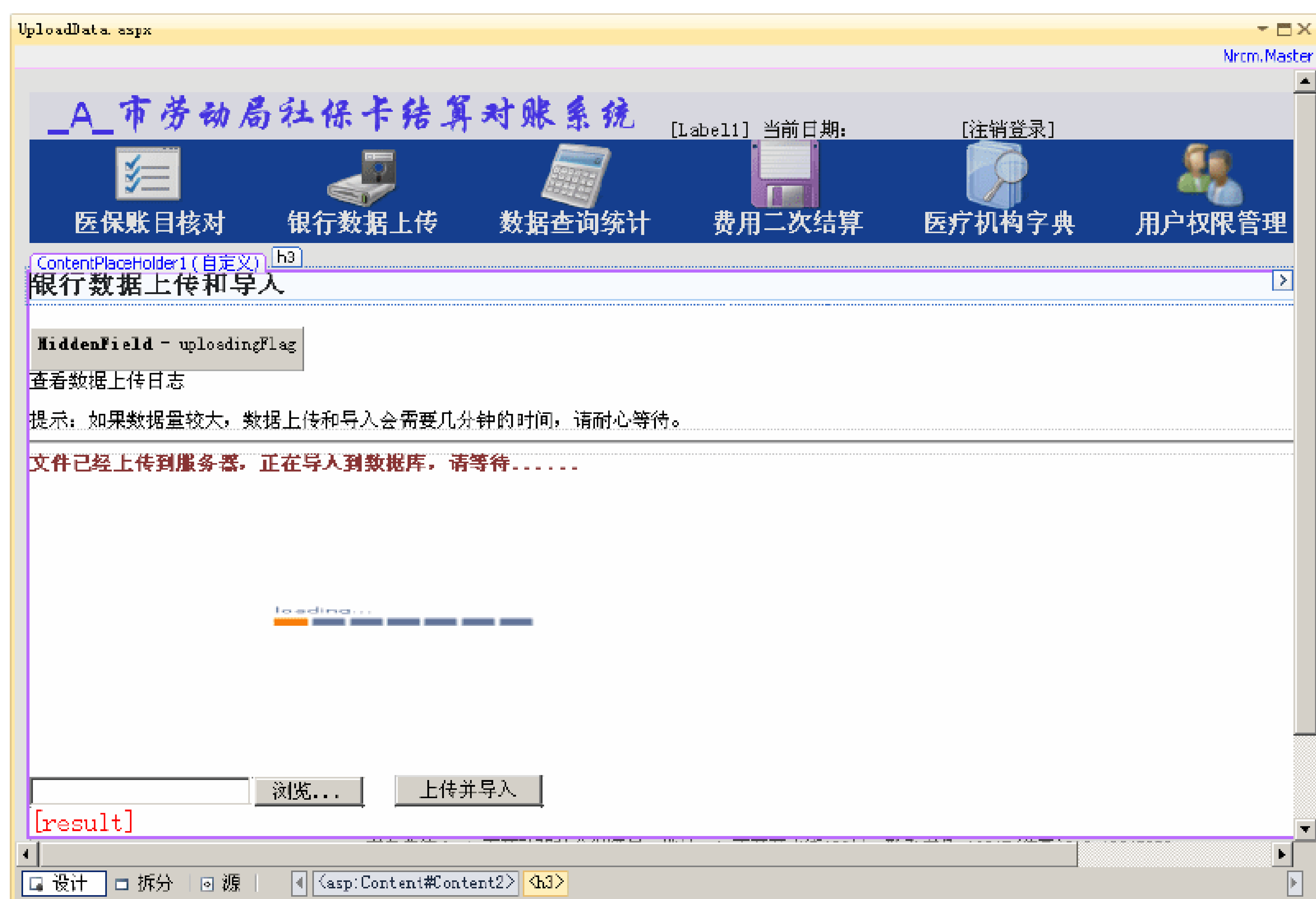


图 13.17　数据上传页面设计视图

```
<h3>银行数据上传和导入</h3>
<asp:HiddenField  runat="server" ID="uploadingFlag" Value="0" /><!--是否有
数据正在上传-->
<br />提示：如果数据量较大，数据上传和导入会需要几分钟的时间，请耐心等待。
 <br /><hr />
<div id="loading" runat="server" visible="false">
<span style="font-weight:bold; color:#833;">
文件已经上传到服务器，正在导入到数据库，请等待......</span><br />
<img src="../images/loading.gif" alt="数据正在处理"  style="height: 200px;
width: 500px"  />
</div>
    <asp:FileUpload ID="FileUpload1" runat="server"  CssClass="upload"/>
       
    <asp:Button ID="Button1" runat="server" Text="上传并导入"  onclick=
    Button1 Click" />
    <br /><!--显示上传结果或者错误提示-->
    <asp:Label ID="result" runat="server" Text="" ForeColor="Red" Font-
    Size="Large"> </asp:Label>
</asp:Content>
```

（3）根据用户需求，本项目只能上传 dbf 文件，在“上传并导入”按钮的客户端 Click 事件中，用 JavaScript 对上传文件类型进行检测。

```
//检测上传文件是否为 dbf 文件
function checkDbf() {
    var file = $('input:file')[0];                          //得到上传控件
    var fname = file.value;                                 //得到上传文件名
    //用正则表达式检查文件名是否以 dbf 结尾，如果不是则提示错误
    if (/.dbf$/.test(fname)) {
        return true;
    }
    else {
        alert('只能上传 dbf 文件。');
        return false;
    }
}
```

（4）在“上传并导入”按钮的服务器端 Click 事件中，编写代码完成数据的上传和导入功能。由于数据文件较大，将其导入 Oracle 数据库需要一定时间，为了使页面尽早返回，使浏览器端能够看到结果，使用一个单独的线程对数据进行导入，并在浏览器端使用 JavaScript 不断刷新数据导入的进度。

```
protected void Button1 Click(object sender, EventArgs e)
{
    //为了防止用户在此次上传未结束前再上传数据，将上传控件隐藏
    FileUpload1.Visible = false;
    Button1.Visible = false;
    uploadingFlag.Value = "1";                          //设置上传标志为 1
    loading.Visible = true;                             //显示上传动画
    string path=Server.MapPath("../upload"+"\\"+FileUpload1.FileName);
    if(File.Exists(path))
        File.Delete(path);
    FileUpload1.SaveAs(path);                           //保存文件
    //创建一个单独的线程进行数据导入
    Thread thread = new Thread(new ParameterizedThreadStart (transfer
    Data));
    thread.Start(path);
}
/// <summary>
/// 将 dbf 文件中的数据导入到 Oracle 数据库
/// </summary>
/// <param name="path">要导入的文件路径</param>
private void transferData(object path)
{
    int success=0, failure=0;                           //导入成功和失败的数目
    bool b = DbfTransferBLL.transferData(path.ToString(),
        out success, out failure);                      //执行数据导入
    UploadLog log = new UploadLog();
    log.userId = WebUtility.currentUser.id;
    log.errorRecord = failure;
    log.correctRecord = success;
    log.totalRecord = failure + success;
    UploadLogBLL.add(log);                              //添加数据上传日志
    //在缓存中添加数据上传状态，浏览器端将通过 JavaScript 调用 Web Service 查询此
    状态
    UploadState state= new UploadState()
    { finished = true, succeededRecord = success,
        failedRecord = failure, totalRecord = failure + success };
    Cache.Insert(Constants.CacheUploadState, state, null,
        DateTime.Now.AddMinutes(5), TimeSpan.Zero);
}
```

上述代码中用到一个类 UploadState，这个类用于描述数据上传状态，类的代码如下。

```
public class UploadState
{
    public bool finished = false;                       //上传和导入是否已经完成
    public int totalRecord=0;                           //dbf 文件中总的记录数
    public int succeededRecord=0;                       //导入成功记录数
    public int failedRecord=0;                          //导入失败记录数
}
```

（5）添加一个 Web 服务，返回数据上传和导入状态，这个 Web 服务被浏览器端通过 AJAX 方式调用。

```
//查询上传数据情况
 [WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
[System.Web.Script.Services.ScriptService]
public class UploadQueryService : System.Web.Services.WebService
{
    static Random r = new Random();
    [WebMethod]
    [ScriptMethod]
    public UploadState refreshUploadState()
    {
        if (Context.Cache[Constants.CacheUploadState] == null)
            return null;
        return Context.Cache[Constants.CacheUploadState] as UploadState;
    }
}
```

（6）在数据上传页面 UploadData.aspx 中，编写 JavaScript 代码以 AJAX 方式不断调用 Web Service 以查询数据导入状态，当数据导入结束后，隐藏动画控件并显示导入结果。

```
<script type="text/javascript">
    var myTimer;                                //定时器，定时刷新数据导入状态
    var label = '#<%=result.ClientID %>';               //上传结果控件 ID
    var progressBar = '#<%=loading.ClientID%>';         //上传动画控件 ID
    var uploadingFlag = '#<%=uploadingFlag.ClientID%>';
                                                        //上传标志控件 ID
    $(function () {
        $('#<%=Button1.ClientID%>').click(checkDbf);    //检测上传文件类型
        var flag = $(uploadingFlag).val();
        if (flag == "1")
        myTimer=setInterval(refreshState,3000);

    });             //$(function())
    //刷新数据上传状态，当数据上传时此方法每隔 3 秒被调用一次
    function refreshState() {
        $.ajax(
        {
            type: "POST",
            data:"{}",
            url:"UploadQueryService.asmx/refreshUploadState",
            dataType: "json",
            contentType: "application/json; charset=utf-8",
            success:getResult
        }
        );
    }
    //取得数据上传的结果
    function getResult(result) {
        if (!result) return;
        if (!result.d) return;
        var r = result.d;
        //数据上传已经完成，则显示上传情况汇总
        if (r.finished) {
            $(uploadingFlag).val("0");                  //清除数据上传标志
            $(progressBar).hide();                      //隐藏上传动画
            if(myTimer)
                clearInterval(myTimer);                 //清除定时器
```

```
                var t = "总共上传数据" + r.totalRecord + "<br/>"
                +"其中成功导入:" + r.succeededRecord+"<br/>"
                + "导入失败：" + r.failedRecord;
                $(label).html(t);
            }
        }
</script>
```

（7）运行 UploadData.aspx 页面，运行界面如图 13.18 所示。



图 13.18　数据上传运行界面

13.5.3　查询数据上传日志

根据用户需求，用户可以根据时间范围查询银行数据上传日志。这个功能较为简单，其中数据访问层和业务逻辑层的功能已经在 13.1 节中实现，本节将介绍表现层 Web 页面的设计和实现。

（1）在表现层项目 AccountCheckWeb 中添加一个页面 UploadLogPage.aspx。

（2）页面上放置两个 TextBox 用于输入日期，一个 Button 用于执行查询，一个 GridView 用于显示查询结果。为了方便用户输入日期，此页面使用 jQuery UI 的 DatePicker 对 ASP.NET 的 TextBox 控件进行扩展，页面代码如下：

```
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <%--导入所需要的 css 文件和 JavaScript 文件--%>
    <link href="../css/ui-lightness/jquery-ui.css" rel="stylesheet"
    type="text/css" />
    <script src="../js/jquery.js" type="text/javascript"></script>
    <script src="../js/jquery-ui.js" type="text/javascript"></script>
    <script src="../js/MyUtility.js" type="text/javascript"></script>
    <script type="text/javascript" language="javascript">
        $(function () {
            //为 gridview 添加光棒效果
            $('table.gridview').find("tr").hover(
            function () { $(this).addClass('hoverRow'); },
```

```
            function () { $(this).removeClass('hoverRow'); }
            ); //$('table').tr.hover
            //为两个日期 TextBox 添加扩展
            $('#<%=date1.ClientID%>').datepicker();
            $('#<%=date2.ClientID%>').datepicker();
        });    //$(function)
    </script>
</asp:Content>
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
<h3>数据上传日志</h3>
时间范围：<asp:TextBox runat="server" id="date1"/>
至<asp:TextBox runat="server" ID="date2" />
<asp:Button runat="server" ID="button1" Text="查询" onclick="button1_
Click" /><br />
<asp:GridView runat="server" ID="grid1" CssClass="gridview" AutoGenerate
Columns="false">
<Columns>
<asp:BoundField HeaderText="上传用户" DataField="UserID" ItemStyle-Width=
"100" />
<asp:BoundField HeaderText="上传日期" DataField="date" ItemStyle-Width=
"150" />
<asp:BoundField HeaderText="总记录数" DataField="TotalRecord" ItemStyle-
Width="120" />
<asp:BoundField  HeaderText=" 导 入 成 功 记 录 数 "  DataField="correctRecord"
ItemStyle-Width="120" />
<asp:BoundField  HeaderText=" 导 入 失 败 记 录 数 "  DataField="errorRecord"
ItemStyle-Width="120" />
</Columns>
</asp:GridView>
<webdiyer:AspNetPager ID="pager" runat="server" onpagechanged="pager
PageChanged">
</webdiyer:AspNetPager>
</asp:Content>
```

（3）在页面后台代码中，为“查询”按钮的 Click 事件和分页控件的 PageChanged 事件编写代码，查询数据并绑定到 GridView 控件中。

```
protected void button1 Click(object sender, EventArgs e)
{
    bindData(true);
}
/// <summary>
/// 将数据上传日志绑定到GridView
/// </summary>
/// <param name="refreshCount">是否需要刷新总记录数（以更新分页控件的总页数）
</param>
private void bindData(bool refreshCount)
{
    //得到要查询的日期范围
    DateRangeArg date = DateRangeArg.between2Date(date1.Text, date2.Text);
    //创建分页查询参数
    PageDataArgument page = new PageDataArgument();
    page.refreshCount = refreshCount;
    page.pageSize = pager.PageSize;
    page.pageIndex = pager.CurrentPageIndex - 1;
    //执行查询，并绑定数据
    var list = UploadLogBLL.getByDate(date, page);
```

```
    if (refreshCount)
        pager.RecordCount = page.count;
    grid1.DataSource = list;
    grid1.DataBind();
}
protected void pager_PageChanged(object sender, EventArgs e)
{
    bindData(false);
}
```

（4）运行 UploadLogPage.aspx 页面，运行界面如图 13.19 所示。



图 13.19　数据上传日志查询页面

13.6　医疗机构对应

在医保卡结算系统中，需要根据不同的医疗机构进行统计和结算。在银行的医保卡消费数据中，使用商户号和终端号来标识不同机构，但是在 A 市人力资源与社会保障局的 Oracle 数据库中，使用社保机构编码和医院编码来标识医疗机构。社保卡结算系统需要在这两种编码之间建立对应关系，才能实现数据共享。

13.6.1　实体类设计

在建立银行商户与医疗机构对应关系时，涉及两个业务实体：银行商户和医疗机构。在业务实体项目 Account.Entity 中添加两个类以描述这两个实体。

```
/// <summary>
/// 医疗机构信息
/// </summary>
public class Hospital
{
    public string InstituteNo { get; set; }                //社保机构编码
```

```
    public string HospitalNo { get; set; }                  //医院编码
    public string Name { get; set; }                        //医院名称
}
/// <summary>
/// 此类用于描述银行数据中的商户信息
/// </summary>
public class BankShop
{
    public string ShopNo { get; set; }                      //商户号
    public string TerminalNo { get; set; }                  //终端号
    public string Name { get; set; }                        //商户名称
    public Hospital CorrespondingHospital { get; set; }    //对应的医疗机构
}
```

13.6.2　数据访问层和业务逻辑层

数据访问层需要实现医疗机构信息查询、商户信息查询，以及建立和修改二者之间的对应关系。商户数据存储在 MD.BANK_TRANSACTION 表中，医疗机构数据存储在 MD.INSTITUTION_NATL 表中，二者对应关系存储在 MD.Institute_Dictionary 表中。为了提高类的内聚性、适当控制类的大小，用 3 个类实现与这 3 个表相关的数据访问。

（1）添加一个 HospitalDB 类，实现医疗机构相关的数据访问功能。代码如下：

```
public static class HospitalDB
{
    #region 数据库表名和列名
    internal const string HospitalCodeColumn = "YYBM";
    internal const string HospitalNameColumn = "YYMC";
    internal const string InstituteCodeColumn = "SBJGBH";
    internal const string HospitalTable = "MD.INSTITUTION_NATL";
    #endregion
    /// <summary>
    /// 根据医院编码和社保机构编码得到医院信息
    /// </summary>
    /// <param name="id">医院编码</param>
    /// <param name="sbjgbh">社保机构编码</param>
    /// <returns>得到的医院数据</returns>
    public static Hospital getByID(string id, string sbjgbh)
    {
        string sql = string.Format("select {0},{1},{4} from {2} where
        {0}='{3}' and {4}='{5}'",
            HospitalCodeColumn, HospitalNameColumn, HospitalTable, id,
            InstituteCodeColumn, sbjgbh);
        var list = getList(sql);
        if (list.Count == 0) return null;
        return list[0];
    }
    /// <summary>
    /// 得到所有医疗机构数据
    /// </summary>
    /// <param name="page">分页参数</param>
    /// <returns>得到的指定页面的医疗机构数据</returns>
    public static List<Hospital> getAllHospitals(PageDataArgument page)
    {
        string sql = string.Format("select * from(select a.*,rownum
```

```
        as rn "
            +"from (select {0},{1},{3} from {2} order by {0}) a) ",
            HospitalCodeColumn, HospitalNameColumn, HospitalTable, Insti
            tuteCodeColumn);
        if (page.refreshCount)
            page.count = MyDbUtility.getCount(sql);
        sql += "where rn between " + page.min + " and " + page.max;
        return getList(sql);
    }
    //根据 select 语句进行查询并返回医院列表
    private static List<Hospital> getList(string select)
    {
        Database oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(select);
        List<Hospital> list = new List<Hospital>();
        using (IDataReader reader = oracle.ExecuteReader(command))
        {
            //循环读取数据，并根据各个字段的值创建 Hospital 对象
            while (reader.Read())
            {
                Hospital entity = new Hospital();
                entity.HospitalNo = reader[HospitalCodeColumn].To-
                String();
                entity.Name = reader[HospitalNameColumn].ToString();
                entity.InstituteNo = reader[InstituteCodeColumn].ToStr-
                ing();
                list.Add(entity);
            }
            return list;
        }
    }
}
```

（2）添加一个 BankShopDB 类，实现与银行商户相关的数据访问功能。

```
public static class BankShopDB
{
    #region 数据库中字段名称
    private const string ShopNoColumn = "CORNO";
    private const string TerminalNoColumn = "TERM";
    private const string NameColumn = "NAME";
    #endregion
    /// <summary>
    /// 取得未建立对应关系的商户列表（没有对应医院的商户）
    /// </summary>
    /// <param name="arg">分页参数</param>
    public static List<BankShop> getUnassignedInstitutes(PageDataArgument
    arg)
    {
        //构建查询语句，是一个嵌套查询和连接查询
        string sql = " select * from "
            +"(select ta.*,rownum as rn from "
            +"( select distinct Corno, term,Name from md.Bank_Transaction
            a "
            +" where not exists "
            + " ( select YYBM from md.institute dictionary  b "
            +" where b.corno=a.corno and b.term=a.term )"
            +" order by corno,term )ta)";
        if (arg.refreshCount)
            arg.count = MyDbUtility.getCount(sql);
```

```
        sql += " where (rn between " + arg.min + " and " + arg.max + ") ";
        return getList(sql);
    }
    /// <summary>
    /// 得到所有商户列表(不管建立和未建立对应关系)
    /// </summary>
    /// <param name="arg">分页参数</param>
    public static List<BankShop> getAllInstitutes(PageDataArgument arg)
    {
        //构建查询用的 select 语句
        string sql = "select * from "
            +"(select a.*,rownum as rn from "
            +"(select distinct Corno, term,Name from md.Bank Transaction "
            +" order by corno,term) a ) ";
        if (arg.refreshCount)
            arg.count = MyDbUtility.getCount(sql);
        sql += " where  rn between " + arg.min + " and " + arg.max;
        //执行查询，得到列表
        List<BankShop> list = getList(sql);
        foreach (BankShop shop in list)
        {
        shop.CorrespondingHospital = InstituteDictionaryDB.getCorres-
        pondingHospital(shop);
        }
        return list;
    }

    /// <summary>
    /// 得到已经建立对应关系的商户列表
    /// </summary>
    /// <param name="arg">分页参数</param>
    /// <returns>查询得到的列表</returns>
    public static List<BankShop> getAssignedInstitutes(PageDataArgument
    arg)
    {
        //构建查询用的 select 语句
        string sql = "select * from "
            +"(select corno,term,yybm,sbjgbh,rownum as rn from institute_
            dictionary) ";
        if (arg.refreshCount)
            arg.count = MyDbUtility.getCount(sql);
        sql += " where (rn between " + arg.min + " and " + arg.max + ")  ";
        Database oracle = MyDbUtility.oracle;
        //创建一个数据库命令
        DbCommand command = oracle.GetSqlStringCommand(sql);
        List<BankShop> list = new List<BankShop>();
        using (IDataReader reader = oracle.ExecuteReader(command))
        {
            //循环读取数据，根据读到的医院编号和银行商户号得到对应的实体对象
            while (reader.Read())
            {
                Hospital hospital = HospitalDB.getByID(reader[2] .ToStr-
                ing(),reader[3].ToString());
                BankShop shop = getBankShopByID(reader[0].ToString(),
                reader[1].ToString());
                if (shop == null) continue;
                shop.CorrespondingHospital = hospital;
                list.Add(shop);
            }
```

```
        }
        return list;
    }
    /// <summary>
    /// 根据商户号和终端号得到商户信息
    /// </summary>
    /// <param name="shop">商户号</param>
    /// <param name="term">终端号 </param>
    public static BankShop getBankShopByID(string shop, string term)
    {
        string sql = string.Format("select Corno, term,Name from md.Bank_
        Transaction "
            + " where corno='{0}' and term='{1}' and rownum=1", shop, term);
        List<BankShop> list = getList(sql);
        if (list.Count > 0)
            return list[0];
        else
            return null;
    }
    /// <summary>
    /// 根据 select 语句查询数据并且创建商户列表
    /// </summary>
    private static List<BankShop> getList(string select)
    {
        Database oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(select);
        List<BankShop> list = new List<BankShop>();
        using (IDataReader reader = oracle.ExecuteReader(command))
        {
            while (reader.Read())
            {
                BankShop entity = new BankShop();
                entity.ShopNo = reader[ShopNoColumn].ToString();
                entity.TerminalNo = reader[TerminalNoColumn].ToString();
                entity.Name = reader[NameColumn].ToString();
                list.Add(entity);
            }
            return list;
        }
    }
}
```

（3）添加一个 InstituteDictionaryDB 类，此类功能为建立和维护银行商户与医疗机构之间的对应关系，并根据对应关系查找另一方。代码如下：

```
public static class InstituteDictionaryDB
{
    #region 数据库表和字段名称
    private const string ShopNoColumn = "CORNO";
    private const string TerminalNoColumn = "TERM";
    private const string HospitalCodeColumn = "YYBM";
    private const string InstituteCodeColumn = "SBJGBH";
    private const string InstituteDictionaryTable="MD.Institute Dictio-
    nary";
    #endregion
    //得到与指定商户对应的医院
    public static Hospital getCorrespondingHospital(BankShop shop)
    {
        Database oracle = MyDbUtility.oracle;
        string sql = string.Format("select {0},{4} from {1} where {2}=:corno
```

```
        and {3}=:term",HospitalCodeColumn,InstituteDictionary Table,
        Shop No Column, TerminalNoColumn,InstituteCodeColumn);
        //构建数据库命令并添加参数
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":corno", DbType.String, shop.Sh-
        opNo);
        oracle.AddInParameter(command, ":term", DbType.String, shop.Ter-
        minalNo);
        //执行命令，读取数据，生成医疗机构列表
        using (IDataReader reader = oracle.ExecuteReader(command))
        {
            if (reader.Read())
            {
                string yybm = reader[0].ToString();
                string jgbh = reader[1].ToString();
                return HospitalDB.getByID(yybm, jgbh);
            }
            else
                return null;
        }
    }
    //得到与医院对应的商户列表
    public static List<BankShop> getCorrespondingShop(string hospital)
    {
        string shopNo = "";
        string terminal = "";
        List<BankShop> list = new List<BankShop>();
        Database oracle = MyDbUtility.oracle;
        //构建查询用的 select 语句
        string sql = string.Format("select {0},{1} from {2} where {3}=:
        yybm",ShopNoColumn, TerminalNoColumn , InstituteDictionary Table,
        HospitalCodeColumn);
        //创建一个命令并添加参数
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":yybm", DbType.String,hospital);
        using (IDataReader reader = oracle.ExecuteReader(command))
        {
            //循环读取数据，创建 BankShop 对象
            while (reader.Read())
            {
                shopNo=reader[0].ToString();
                terminal = reader[1].ToString();
                BankShop shop = new BankShop()
                    { ShopNo = shopNo, TerminalNo = terminal };
                list.Add(shop);
            }
        }
        return list;
    }
    /// <summary>
    /// 保存医疗机构与商户之间的对应关系
    /// </summary>
    /// <param name="shop">商户对象（包含其对应医院）</param>
    public static int saveRelation(BankShop shop)
    {
        Database oracle = MyDbUtility.oracle;
        //先删除原有对应关系
        string sql = string.Format("delete from {0} where {1}=:corno and
```

```
        {2}=:term",
            InstituteDictionaryTable,ShopNoColumn, TerminalNoColumn);
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":corno", DbType.String, shop.
        ShopNo);
        oracle.AddInParameter(command, ":term", DbType.String, shop.Term
        inalNo);
        oracle.ExecuteNonQuery(command);
        //创建新的对应关系
        sql=string.Format(" insert into {0} ({1},{2},{3},{4}) values
            (:corno,:term,:yybm,:sbjgbh)", InstituteDictionaryTable,
            ShopNoColumn, TerminalNoColumn, HospitalCodeColumn, Institu-
            teCodeColumn);
        command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":corno", DbType.String, shop.
        ShopNo);
        oracle.AddInParameter(command, ":term", DbType.String, shop.
        TerminalNo);
        oracle.AddInParameter(command, ":yybm", DbType.String, shop.
        CorrespondingHospital.HospitalNo);
        oracle.AddInParameter(command, ":sbjgbh", DbType.String, shop.
        CorrespondingHospital.InstituteNo);
        return oracle.ExecuteNonQuery(command);
    }
    /// <summary>
    /// 删除指定商户的对应关系
    /// </summary>
    /// <param name="corno">商户号</param>
    /// <param name="term">终端号</param>
    public static int deleteRelation(string corno, string term)
    {
        Database oracle = MyDbUtility.oracle;
        string sql = string.Format("delete from {0} where {1}=:corno and
        {2}=:term", InstituteDictionaryTable, ShopNoColumn, TerminalNoCo-
        lumn);
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":corno", DbType.String, corno);
        oracle.AddInParameter(command, ":term", DbType.String, term);
        return oracle.ExecuteNonQuery(command);
    }
    /// <summary>
    /// 自动对应功能，在名称相同的医疗机构和银行商户之间建立对应关系
    /// </summary>
    public static int autoCorrespond()
    {
        string sql =
            @"insert into institute_dictionary(corno,term,yybm,sbjgbh)
                select corno,term,yybm from
                (
                select distinct b.corno,b.term,a.yybm,a.sbjgbh
                from institution_natl a inner join bank_transaction b
                on a.yymc = b.name
                where not exists
                (select * from institute_dictionary i
                where i.corno=b.corno and i.term=b.term)   )";
        Database oracle = MyDbUtility.oracle;
        DbCommand command=oracle.GetSqlStringCommand(sql);
```

```
            return oracle.ExecuteNonQuery(command);
        }
}
```

（4）业务逻辑层较为简单，其功能为调用数据访问层的相应方法，此处省略其代码。

13.6.3　医疗机构对应页面

在医疗机构对应页面中，用户可以建立、查看和修改银行商户与医疗机构之间的对应关系。页面整体分为两个区域，上面的区域用于显示和删除对应关系，下面的区域用于创建对应关系，在某一时刻这两个区域仅有一个可见。页面设计视图如图 13.20 所示。

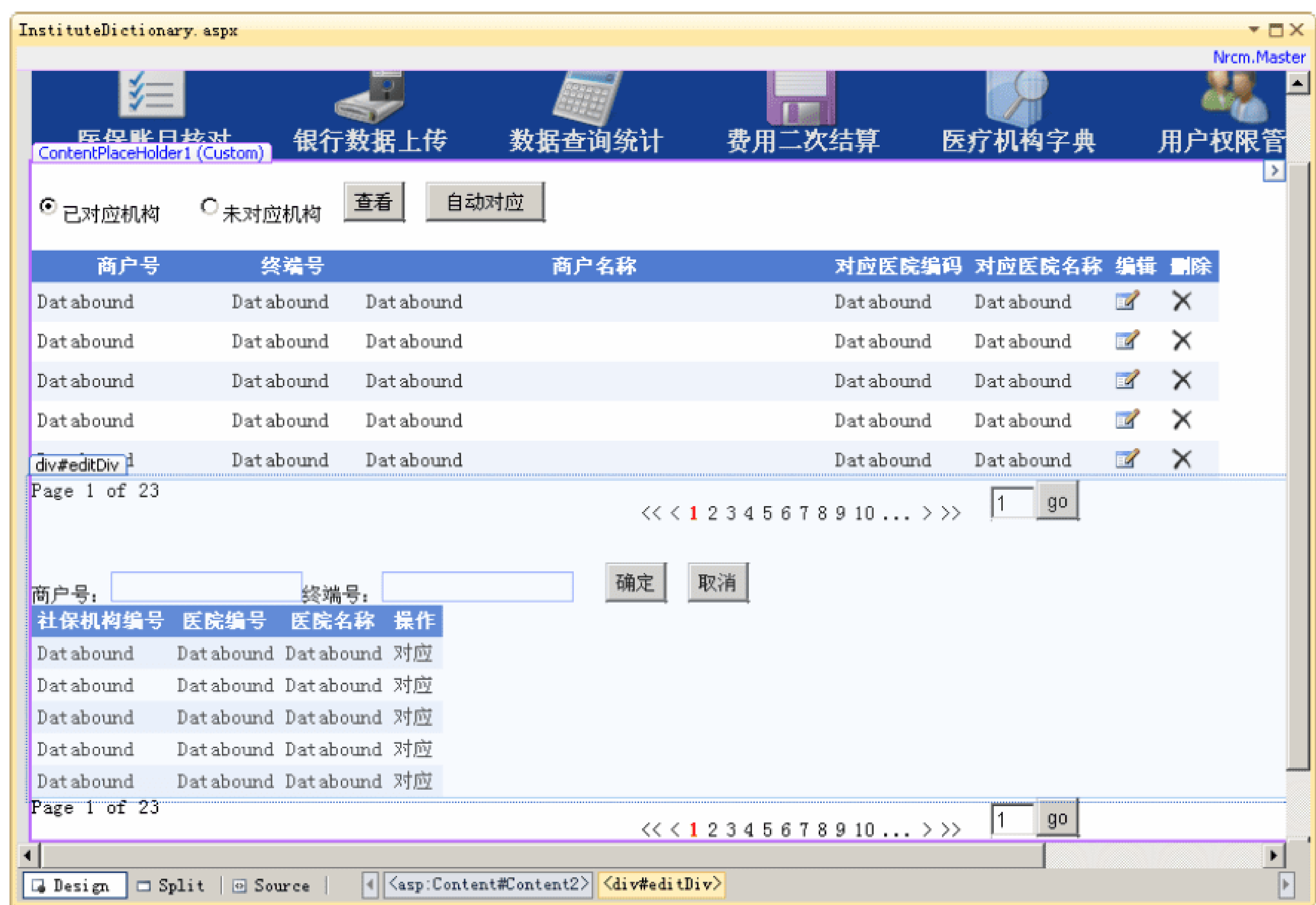


图 13.20　医疗机构对应页面设计视图

（1）在表现层项目 AccountCheckWeb 中添加一个页面 InstituteDictionary.aspx，页面布局如图 13.20 所示，页面代码如下：

```
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
<div id="tabpage1" runat="server"><!--这个 div 为显示对应关系的区域--> <br />
    <asp:RadioButton ID="radio1" runat="server" Text="已对应机构" Group-
    Name="group1" Checked="true"/>  
    <asp:RadioButton ID="radio2" runat="server" Text="未对应机构" Group-
    Name="group1" />  <asp:Button
        ID="Button1" runat="server" Text="查看" onclick="Button1_Click"
        /> 
    <asp:Button   ID="Button2" runat="server" Text="自动对应" onclick=
    "Button2 Click" Visible="false"  />
    <br /><br />
    <!--显示数据的 GridView-->
    <asp:GridView ID="grid" runat="server" CssClass="gridview"
        AutoGenerateColumns="False" DataKeyNames="ShopNo,TerminalNo"
        onrowdeleting="grid_RowDeleting" onrowediting="grid_RowEditing">
        <Columns>
```

```
            <asp:BoundField DataField="ShopNo" HeaderText="商户号">
            <HeaderStyle Width="120px" />
            </asp:BoundField>
            <asp:BoundField DataField="TerminalNo" HeaderText="终端号">
            <HeaderStyle Width="80px" />
            </asp:BoundField>
            <asp:BoundField DataField="Name" HeaderText="商户名称">
            <HeaderStyle Width="300px" />
            </asp:BoundField>
            <asp:TemplateField HeaderText="对应医院编码">
            <ItemTemplate>
            <%#Eval("CorrespondingHospital.HospitalNo")%>
            </ItemTemplate>
            </asp:TemplateField>
            <asp:TemplateField HeaderText="对应医院名称">
            <ItemTemplate>
            <%#Eval("CorrespondingHospital.Name")%>
            </ItemTemplate>
            </asp:TemplateField>
            <asp:CommandField EditImageUrl="~/images/edit.png" HeaderText="
            编辑"
            EditText="" ShowEditButton="true" ButtonType="Image"  />
            <asp:CommandField DeleteImageUrl="~/images/delete.png" Header-
            Text="删除"
                DeleteText="" ShowDeleteButton="True" ButtonType="Image" />
        </Columns>
    </asp:GridView>
    <webdiyer:AspNetPager ID="pager1" runat="server"
            onpagechanged="pager1_PageChanged" ShowCustomInfoSection=
            "Left"
            ShowBoxThreshold="10">
    </webdiyer:AspNetPager>
</div> <!--tabpage1 结束-->
<!--这个 div 为编辑和创建对应关系的区域-->
<div id="editDiv" runat="server"><br /><br />
    商户号：<asp:TextBox ID="shop" runat="server" ></asp:TextBox>
    终端号：<asp:TextBox ID="terminal" runat="server"></asp: TextBox>
      
    <asp:Button ID="OK" runat="server" Text="确定" onclick="OK Click"
    Visible="false" />  
    <asp:Button ID="cancel" runat="server" Text="取消" onclick="cancel
    Click"  /><br />
    <!--显示医院列表-->
    <asp:GridView ID="hospitalGrid" runat="server" AutoGenerateColumns=
    "False"
        DataKeyNames="HospitalNo" CssClass="gridview"
        onselectedindexchanged="hospitalGrid_SelectedIndexChanged">
        <Columns>
            <asp:BoundField DataField="InstituteNo" HeaderText="社保机构编号
             " />
            <asp:BoundField DataField="HospitalNo" HeaderText="医院编号" />
            <asp:BoundField DataField="Name" HeaderText="医院名称" />
            <asp:CommandField ShowSelectButton="True" SelectText="对应"
            HeaderText="操作" />
        </Columns>
    </asp:GridView>
    <webdiyer:AspNetPager ID="pager2" runat="server" ShowCustomInfo-
    Section="Left"
```

```
        ShowBoxThreshold="10" onpagechanged="pager2_PageChanged">
    </webdiyer:AspNetPager>
</div>
</asp:Content>
```

（2）在 Page_Load 事件中，加载第一页银行商户数据和医疗机构数据。

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        loadShop(true);
        loadHospital(true);
    }
}
/// <summary>
/// 加载一页医院数据
/// </summary>
/// <param name="firstTime">是否第一次加载（首次加载时刷新总记录数）</param>
private void loadHospital(bool firstTime)
{
    //创建分页检索参数并设置各个属性
    PageDataArgument page = new PageDataArgument();
    page.pageSize = pager2.PageSize;
    page.pageIndex = pager2.CurrentPageIndex - 1;
    page.refreshCount = firstTime;
    //执行查询，并绑定数据
    List<Hospital> list = HospitalBLL.getAllHospitals(page);
    pager2.RecordCount = page.count;
    hospitalGrid.DataSource = list;
    hospitalGrid.DataBind();
}
/// <summary>
/// 加载一页商户数据
/// </summary>
/// <param name="firstTime">是否第一次加载（首次加载时刷新总记录数）</param>
private void loadShop(bool firstTime)
{
    List<BankShop> list=null;
    //创建分页检索参数并设置各个属性
    PageDataArgument arg = new PageDataArgument();
    arg.pageSize = pager2.PageSize;
    arg.pageIndex = pager2.CurrentPageIndex - 1;
    arg.refreshCount = firstTime;
    if(firstTime)pager1.CurrentPageIndex = 1;
    //加载已对应的商户还是未对应的商户
    if(radio1.Checked)
        list=BankShopBLL.getAssignedInstitutes(arg);
    else
        list = BankShopBLL.getUnassignedInstitutes(arg);
    if (firstTime)
    {
        pager1.RecordCount = arg.count;
    }
```

```
    grid.DataSource = list;
    grid.DataBind();
}
```

（3）在两个分页控件的 PageChanged 事件中，加载新的一页数据。

```
protected void pager1 PageChanged(object sender, EventArgs e)
{
    loadShop(false);
}
protected void pager2 PageChanged(object sender, EventArgs e)
{
    loadHospital(false);
}
```

（4）在第一个 GridView 的 RowDeleting 事件中，删除指定的对应关系。

```
protected void grid RowDeleting(object sender, GridViewDeleteEventArgs e)
{
    string shopno = grid.Rows[e.RowIndex].Cells[0].Text;
    string term = grid.Rows[e.RowIndex].Cells[1].Text;
    InstituteDictionaryBLL.deleteRelation(shopno, term);
    loadShop(false);
}
```

（5）在第一个 GridView 的 RowEditing 事件中，进入编辑视图，并根据当前行设置商户号和终端号。

```
protected void grid RowEditing(object sender, GridViewEditEventArgs e)
{
   editDiv.Visible = true;
   tabpage1.Visible = false;
   shop.Text = grid.DataKeys[e.NewEditIndex][0].ToString();
   terminal.Text = grid.DataKeys[e.NewEditIndex][1].ToString();
   e.Cancel = true;                            //取消编辑事件
}
```

（6）在第二个 GridView（医院列表）的 SelectedIndexChanged 中，在所选中的医院和当前商户之间建立对应关系。

```
protected void hospitalGrid SelectedIndexChanged(object sender, Event-
Args e)
{
   BankShop bankshop = new BankShop();
   Hospital h= new Hospital();
   h.HospitalNo = hospitalGrid.SelectedRow.Cells[1].Text; //得到医院编码
   h.InstituteNo= hospitalGrid.SelectedRow.Cells[0].Text;
                                                   //得到社保机构编码
   bankshop.CorrespondingHospital = h;
   bankshop.ShopNo = shop.Text;
   bankshop.TerminalNo = terminal.Text;
   int n = InstituteDictionaryBLL.saveRelation(bankshop);
                                                   //建立对应关系
   loadShop(false);
}
```

（7）运行 InstitueDictionary.aspx 页面，运行界面如图 13.21 所示。



图 13.21　医疗机构对应页面

13.7　账　目　核　对

各个医疗机构每隔一段时间需要到 A 市人力资源与社会保障局进行账目核对和资金结算，以银行提供的数据以依据，统计在本机构产生的消费数据，并获得相应的资金。

13.7.1　数据访问层和业务逻辑层

A 市人力资源与社会保障局有一套社保管理系统，其中包含参保人员在医疗机构的消费数据。另外，如前所述，银行也向 A 市人力资源与社会保障局提供一套由银行系统产生的社保卡消费数据。从理论上来说，这两套数据应该一致。本系统需要对这两套数据进行核对，如果存在不一致的数据，则显示出来以方便用户查找原因并改正。

由于银行数据 MD.Bank_Transaction 表中数据量很大，直接在这个表上进行查询和统计会花费较多时间。为了提高性能，可以将待统计的原始数据（通常为某一医院某时间范围内的数据）复制到一个临时表中，然后再对临时表中的数据进行查询和统计。为了避免冲突，每个用户所使用的临时表名应该各不相同。

在数据访问层项目 AccountCheckDAL 中添加一个类 AccountCheckDB，代码如下：

```
public class AccountCheckDB
{
    #region Fields
    private string tableName;                          //临时表名称
    private CheckAccountArg checkArg;                  //核对账目参数
    #endregion
    #region Constants
    //创建临时表的 SQL 语句
    private const string CreateTableSql1 = "CREATE TABLE ";
    private const string CreateTableSql2=@" (ID CHAR(20) NOT NULL PRIMARY
```

```
KEY ENABLE,
CORNO VARCHAR2(30),  NAME VARCHAR2(60), ACCNO VARCHAR2(40),
AMT NUMBER, FEE NUMBER, REALAMT NUMBER,  TYPE VARCHAR2(20),
TERM VARCHAR2(20), TIME DATE,  CSERNO VARCHAR2(20),
YYBM varchar2(18), SBJGBH varchar2(10), SFZHM VARCHAR2(18),
FSSJ DATE, JYJE NUMBER, XM varchar2(20), ERROR_FLAG char(1),
BRQX varchar2(50))";
private const string TemplateTableName = "TEMP BANK CHECK ";
                                                    //临时表名称前缀
#endregion
#region Public Methods
//进行账目核对
public void check(CheckAccountArg arg)
{
    if (string.IsNullOrEmpty(arg.user))
        throw new ArgumentNullException("用户 ID");
    if(string.IsNullOrEmpty(arg.hospital))
        throw new ArgumentNullException("医院编码");
    if (string.IsNullOrEmpty(arg.institute))
        throw new ArgumentNullException("社保机构编号");
    checkArg = arg;
    tableName = getTableName(arg.user) ;
    initTable();                              //创建临时表
    initReportData();                         //初始化临时表数据
    //processReportData();
    //getHosptialData();
    getPersonInfo();                          //得到人员信息
}
//删除临时表
public static void deleteTempTables(string user)
{
    if (string.IsNullOrEmpty(user))
        throw new ArgumentNullException("user");
    MyDbUtility.dropTableIfExists(getTableName(user));
}
//根据用户名得到临时表名称
public static string getTableName(string user)
{
    return TemplateTableName+user;
}
#endregion
#region private methods
/// <summary>
/// 删除再重建临时表以清除表中数据
/// </summary>
private void initTable()
{
    MyDbUtility.dropTableIfExists(tableName);
    string sql = CreateTableSql1 + tableName + CreateTableSql2;
    Database oracle = MyDbUtility.oracle;
    oracle.ExecuteNonQuery(CommandType.Text, sql);
}
/// <summary>
/// 将数据从 MD.BANK_TRANSACTION 中复制到临时表中
/// </summary>
private void initReportData()
{
    //构建插入语句
    string insert = "insert into "+tableName
```

```
            + "(id,corno,name,accno,amt,fee,realamt,type,"
            +" term,time,cserno,ERROR FLAG,yybm,sbjgbh)"
            +" select id,corno,name,accno,amt,fee,realamt,"
            +" type,term,time,cserno,'0',:yybm,:sbjgbh "
            +" from md.bank_transaction "
            +" where audit_no is null and corno=:corno and term=:term "
            +" and time between :begin and :end";
        Database oracle = MyDbUtility.oracle;
        //创建一个命令对象并添加各个参数
        DbCommand command = oracle.GetSqlStringCommand(insert);
        oracle.AddInParameter(command, ":yybm", DbType.String,checkArg.
        hospital);
        oracle.AddInParameter(command, ":sbjgbh", DbType.String,checkArg.
        institute);
        oracle.AddInParameter(command, ":corno", DbType.String);
        oracle.AddInParameter(command, ":term", DbType.String);
        oracle.AddInParameter(command, ":begin", DbType.DateTime,check-
        Arg.beginDate);
        oracle.AddInParameter(command, ":end", DbType.DateTime,checkArg.
        endDate);
        List<BankShop> list=InstituteDictionaryDB.getCorrespondingShop
        (checkArg.hospital);
        //将与指定医疗机构相对应的商户消费记录导入临时表
        foreach (BankShop item in list)
        {
            oracle.SetParameterValue(command, ":corno", item.ShopNo);
            oracle.SetParameterValue(command, ":term", item.TerminalNo);
            int n=oracle.ExecuteNonQuery(command);
        }
    }
    /// <summary>
    /// 删除相互抵消的数据，包括[PCA-PCC], [PPT-PPC], [PCA-PPT]
    /// </summary>
    /// <remarks>
    ///  PCA 正常消费，PPT 退货，PCC 消费冲正，PPC 退货冲正
    /// </remarks>
    public void processReportData()
    {
        string[][] pairs={
              new string[]{"PCA","PCC"},
              new string[]{"PPT","PPC"},
              new string[]{"PCA","PPT"}};
        Database oracle = MyDbUtility.oracle;
        //先将数据插入到一个临时表中
        string sql="insert into temp id table (id1,id2) select a.id as
        id1,b.id  as id2 from "
                +tableName  +" a, "+tableName
            +" b where a.cserno =b.cserno and upper(a.type)=:type1 and
            upper(b.type)=:type2";
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":type1", DbType.String);
        oracle.AddInParameter(command, ":type2", DbType.String);
        for (int i = 0; i < pairs.Length; i++)
        {
            //先删除临时表，再重新创建临时表，以清空数据
            MyDbUtility.dropTableIfExists("temp_id_table");
            sql="create table temp id table (id1 varchar2(20),id2
            varchar2(20))";
            oracle.ExecuteNonQuery(CommandType.Text,sql);
            string a = pairs[i][0];
```

```
            string b = pairs[i][1];
            oracle.SetParameterValue(command, ":type1", a);
            oracle.SetParameterValue(command, ":type2", b);
            oracle.ExecuteNonQuery(command);
            //删除临时表中存在的互相抵消的数据
            sql = "delete from " + tableName + " where id in (select id1 from
            temp_id_table)";
            oracle.ExecuteNonQuery(CommandType.Text, sql);
            sql = "delete from " + tableName + " where id in (select id2 from
            temp_id_table)";
            oracle.ExecuteNonQuery(CommandType.Text, sql);
        }
        MyDbUtility.dropTableIfExists("temp_id_table");
    }
    //设置临时表中医疗机构数据
    public void getHosptialData()
    {
        Database oracle = MyDbUtility.oracle;
        string sql = "update " + tableName + " set yybm =:yybm,sbjgbh=
        :sbjgbh";
        DbCommand command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":yybm", DbType.String, checkArg.
        hospital);
        oracle.AddInParameter(command, ":sbjgbh", DbType.String, check-
        Arg.institute);
        oracle.ExecuteNonQuery(command);
        getHospitalMoney();
    }
    //设置参保人员信息
    private void getPersonInfo()
    {
        Database oracle = MyDbUtility.oracle;
        DbCommand command = null;
        string sql = null;
        //设置身份证号
        sql = "update " + tableName +
            " a set sfzhm=(select sfzhm from dis.card_payout b where rownum=1
            and a.accno=b.kyhzh)";
        command = oracle.GetSqlStringCommand(sql);
        oracle.ExecuteNonQuery(command);
        //设置姓名
        sql = "update " + tableName + " a set xm=(select xm from md.emp_natl
        b where rownum=1 and a.sfzhm=b.grbh)";
        command = oracle.GetSqlStringCommand(sql);
        oracle.ExecuteNonQuery(command);
        //设置所属区县
        sql="update "+tableName+" temp set brqx = "
            +"(select ssqx from md.card_account card,md.sbjgqx qx "
            +"where  card.kyhzh=temp.accno and qx.sbjgbh=card.sbjgbh and
            rownum=1)";
        command = oracle.GetSqlStringCommand(sql);
        oracle.ExecuteNonQuery(command);
    }
    //获得社保管理系统中的消费金额
    private void getHospitalMoney()
    {
        Database oracle = MyDbUtility.oracle;
        DbCommand command = null;
        string sql = null;
        //修改发生时间和交易金额
```

```
            sql = "update " + tableName
                  + " a set (fssj,jyje) =( select fssj,jyje from dis.card payout
                  b "
                  +" where b.yybm=:yybm and b.sbjgbh=:sbjgbh "
                  +" and substr(cserno,length(a.cserno)-5,6)=substr(yhjyh,
                  length (b.yhjyh)-5,6) "
                  +"and a.accno=b.KYHZH)";
            command = oracle.GetSqlStringCommand(sql);
            oracle.AddInParameter(command, ":yybm", DbType.String, checkArg.
            hospital);
            oracle.AddInParameter(command, ":sbjgbh", DbType.String, check
            Arg.institute);
            oracle.ExecuteNonQuery(command);
            //设置错误标记
            sql = "update " + tableName + @" set error flag ='1' where
                  fssj<time+(-10/(24*60)) or fssj>time+(10/(24*60))
                  or nvl(jyje,0)<amt - 0.01 or nvl(jyje,0)>amt+0.01";
            oracle.ExecuteNonQuery(CommandType.Text, sql);
        }
        #endregion
}
//对账数据检索参数
public class CheckAccountArg
{
    public string user;                             //查询用户 ID
    public string hospital;                         //医疗编码
    public string institute;                        //社保机构编码
    public DateTime beginDate;                      //查询开始日期
    public DateTime endDate;                        //查询结束日期
}
```

13.7.2　对账页面

在对账页面中，用户可以选择一个医疗机构，查看某一时间范围内的社保卡消费数据，并将社保管理系统数据与银行数据进行核对。对账页面 CheckAccount2.aspx 设计外观如图 13.22 所示。



图 13.22　社保卡对账页面设计外观

在图 13.22 所示的社保卡对账页面中，可以从下拉列表中选择一个医疗机构，输入一个日期范围，查看满足条件的社保卡消费数据。由于医疗机构数量众多，为了能够快速从下拉列表中找到特定医院，在下拉列表旁边增加了根据医院编码进行筛选的功能。根据用户需求，用户只可以查看拥有权限的医疗机构的数据，所以在绑定医疗机构下拉列列表时要根据用户权限进行过滤。

（1）在表现层项目 CheckAccountWeb 项目中添加一个页面 CheckAccount2.aspx，页面代码如下：

```
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
   <script type="text/javascript">
      var d1 = '#<%=date1.ClientID %>';           //开始日期控件 ID
      var d2 = '#<%=date2.ClientID %>';           //终止日期控件 ID
      $(init);
      function init() {
         $('table.gridview').find("tr").hover(
         function() { $(this).addClass('hoverRow'); },
         function() { $(this).removeClass('hoverRow'); }
         ); //$('table').tr.hover
         $SjlUtility.addButtonClass();
         $SjlUtility.jQueryDatePickerChinese();
         createDatePicker();
         //为 3 个打印按钮添加事件处理程序
         $('#printReport').click(printReport);
         $('#printDetail').click(printDetail);
         $('#countyButton').click(printCounty);
      };
      //将两个日期文本框扩展成 jQuery 日历控件
      function createDatePicker() {
         $(d1).datepicker();
         $(d2).datepicker();
      };
      //检测用户输入的日期
      function checkInput() {
         if ($(d1).val() == "" || $(d2).val() == "") {
            alert('必须选择日期! ');
            return false;
         }
         return true;
      };
      //打开汇总报表窗口
      function printReport() {
         window.showModalDialog("PrintCheckResult.aspx",null,
         "dialogWidth=600px;dialogHeight=450px;");
      };
      //打开明细报表窗口
      function printDetail() {
         window.showModalDialog("PrintDetail.aspx", null,
         "dialogWidth=800px;dialogHeight=600px;");
      }
      //打开区县汇总报表窗口
      function printCounty() {
         window.showModalDialog("PrintCountySummary.aspx", null,
         "dialogWidth=600px;dialogHeight=450px;");
      }
   </script>
</asp:Content>
```

```
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
    <br />医疗机构：<asp:DropDownList ID="hospitalList" runat="server">
    </asp:DropDownList>
    根据医院编码筛选：<asp:TextBox ID="searchWord" runat="server">
    </asp:TextBox>
    <asp:Button  ID="filter" runat="server" Text="筛选" onclick="filter
    Click" /> <br />
    日期：<asp:TextBox ID="date1" runat="server"></asp:TextBox>
    至<asp:TextBox ID="date2" runat="server"></asp:TextBox>
    <asp:Button ID="OK"  runat="server" Text="核对账目" OnClientClick=
    "return checkInput();" onclick="OK Click" />
    <input type="button" id="printReport" value="汇总打印" />
    <input type="button" id="countyButton" value="区县汇总" />
    <input type="button" id="printDetail" value="明细打印" /><br />
<div>
    <asp:GridView ID="grid1" runat="server" CssClass="gridview" Auto
    GenerateColumns="true">
        <EmptyDataTemplate>
            <span style="color:Red; font-weight:bold;">没有符合条件的数据
            </span>
        </EmptyDataTemplate>
    </asp:GridView>
    <webdiyer:AspNetPager ID="pager1" runat="server" ShowBoxThreshold="10"
        onpagechanged="pager1 PageChanged" >
    </webdiyer:AspNetPager>
    <div>
          总计<asp:Label ID="transactionCount" runat="server" Text="0">
          </asp:Label>人次，
          费用总额：<asp:Label ID="totalMoney" runat="server" Text="0">
          </asp:Label>，
          银联手续费：<asp:Label ID="totalFee" runat="server" Text="0">
          </asp:Label>，
          应拨付金额：<asp:Label ID="realMoney" runat="server" Text="0">
          </asp:Label>
   </div>
</div>
</asp:Content>
```

（2）在 AccountCheck2.aspx 的 Page_Load 事件中，在医疗机构下拉列表中加载当前登录用户有权访问的所有医疗机构。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindHospitalList();
    }
}
/// <summary>
/// 根据用户权限绑定医院列表
/// </summary>
private void bindHospitalList()
{
    hospitalList.Items.Clear();
    //如果当前用户是医疗机构用户则只可以查看自己的数据
    if (Common.WebUtility.isMedicineShopUser())
    {
```

```
            filter.Visible = false;
           Hospital h = HospitalBLL.getByID(Common.WebUtility.currentUser.id,
          AccountCheckWeb.Common.WebUtility.instituteId);
            ListItem item = new ListItem(h.Name, h.HospitalNo+","+
            h.InstituteNo);
            hospitalList.Items.Add(item);
        }
        else
        {
            //如果当前用户不是医疗机构用户而是社会保障局用户，根据分配的医院权限填充医院
              列表
            WebUser user=WebUtility.currentUser;
            List<Hospital> list = HospitalRightBLL.getGrantedHospitals
             (user.id,
                searchWord.Text.Trim(), PageDataArgument.all);
            for (int i=0;i<list.Count;i++)
            {
                Hospital h = list[i];
                if (h == null) return;
                ListItem item = new ListItem(h.Name, h.HospitalNo + "," + h.
                InstituteNo);
                hospitalList.Items.Add(item);
            }
        }
}
```

（3）在“核对账目”按钮的 Click 事件中，核对满足指定条件的数据，并将结果显示在 GridView 中。

```
protected void OK Click(object sender, EventArgs e)
{
    //检测用户输入合法性
    DateTime d1 = DateTime.Parse(date1.Text);
    DateTime d2 = DateTime.Parse(date2.Text).AddDays(1).AddSeconds(-1);
    if (d1 > d2) return;
    string[] temp = hospitalList.SelectedValue.Split(',');
    //生成对账数据查询参数
    CheckAccountArg arg=new CheckAccountArg()
    {
        user = Common.WebUtility.currentUser.id,
        hospital = temp[0],
        institute = temp[1],
        beginDate=d1,endDate=d2
    };
    Session[Common.Constants.SessionCheckArg] = arg;
                                    //将查询参数保存到 Session
    AccountCheckBLL.AccountCheckBLL bll =
        new
AccountCheckBLL.AccountCheckBLL(Common.WebUtility.currentUser.id);
    bll.check(arg);                 //生成对账数据
    //在页面底部显示汇总数据
    CheckAccountReport report = bll.getSummary();
    totalMoney.Text = string.Format("{0:C}", report.amt);
    totalFee.Text = string.Format("{0:C}", report.fee);
    realMoney.Text = string.Format("{0:C}", report.realamt);
    transactionCount.Text = report.count.ToString();
    loadData(true);
}
/// <summary>
```

```
/// 加载要对账的数据并显示在 GridView 中
/// </summary>
/// <param name="firstTime">是否首次加载（首次加载需要刷新记录总数）</param>
private void loadData(bool firstTime)
{
    AccountCheckBLL.AccountCheckBLL bll =
        new AccountCheckBLL.AccountCheckBLL(Common.WebUtility.
        currentUser.id);
    PageDataArgument arg = new PageDataArgument();
    arg.pageIndex=pager1.CurrentPageIndex - 1;
    arg.pageSize=pager1.PageSize;
    arg.refreshCount=firstTime;
    DataTable table = bll.getAllData( arg);
    if (firstTime)
        pager1.RecordCount = arg.count;
    grid1.DataSource = table;
    grid1.DataBind();
}
```

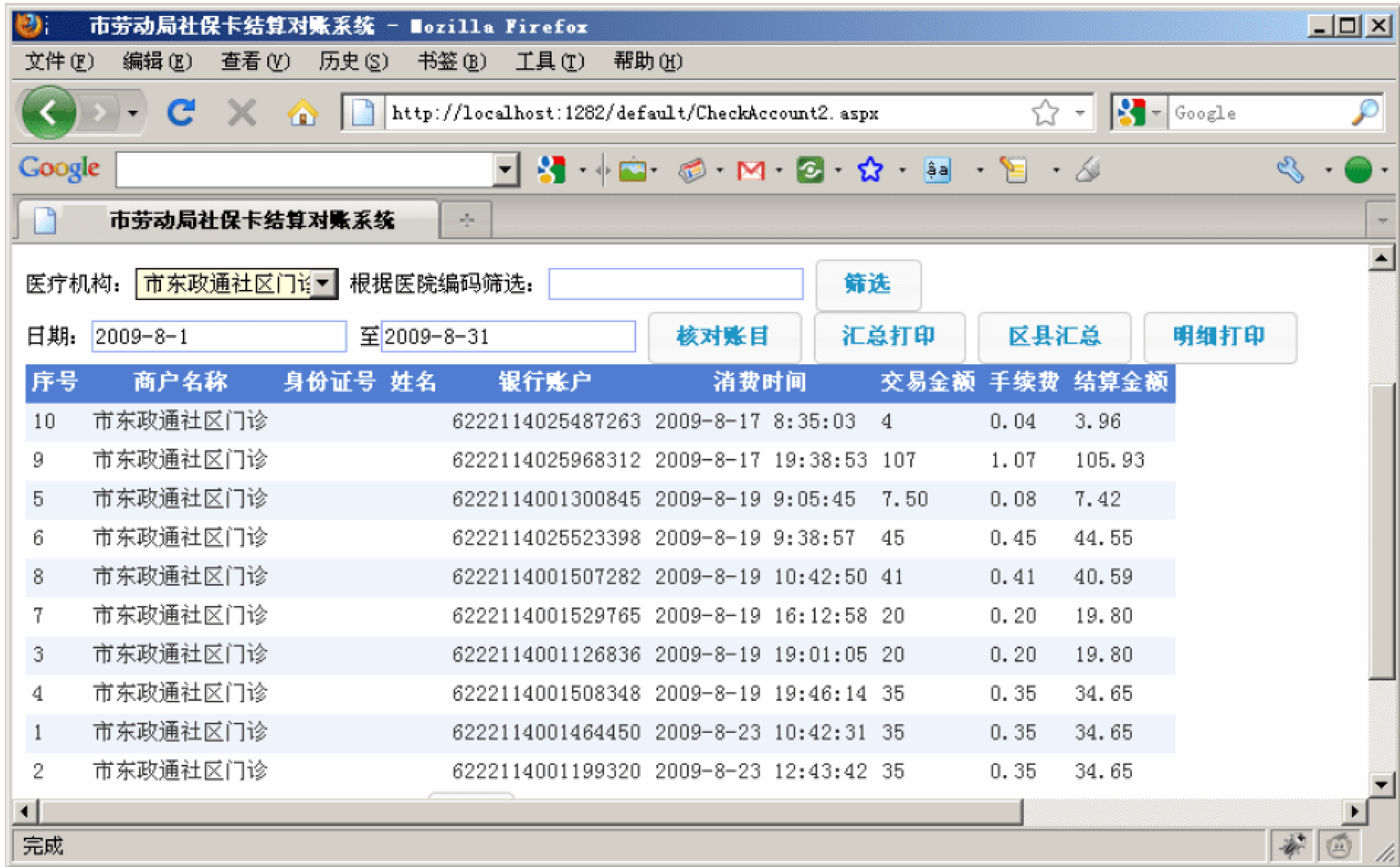
（4）在分页控件的 PageChanged 事件中，绑定新的一页数据。

```
protected void pager1_PageChanged(object sender, EventArgs e)
{
    loadData(false);
}
```

（5）在“筛选”按钮的 Click 事件中，根据用户输入的医院编码对医疗机构列表进行筛选，并重新绑定医院列表。

```
protected void filter Click(object sender, EventArgs e)
{
    bindHospitalList();
}
```

（6）运行 AccountCheck2.aspx 页面，运行结果如图 13.23 所示。



序号	商户名称	身份证号	姓名	银行账户	消费时间	交易金额	手续费	结算金额
10	市东政通社区门诊			6222114025487263	2009-8-17 8:35:03	4	0.04	3.96
9	市东政通社区门诊			6222114025968312	2009-8-17 19:38:53	107	1.07	105.93
5	市东政通社区门诊			6222114001300845	2009-8-19 9:05:45	7.50	0.08	7.42
6	市东政通社区门诊			6222114025523398	2009-8-19 9:38:57	45	0.45	44.55
8	市东政通社区门诊			6222114001507282	2009-8-19 10:42:50	41	0.41	40.59
7	市东政通社区门诊			6222114001529765	2009-8-19 16:12:58	20	0.20	19.80
3	市东政通社区门诊			6222114001126836	2009-8-19 19:01:05	20	0.20	19.80
4	市东政通社区门诊			6222114001508348	2009-8-19 19:46:14	35	0.35	34.65
1	市东政通社区门诊			6222114001464450	2009-8-23 10:42:31	35	0.35	34.65
2	市东政通社区门诊			6222114001199320	2009-8-23 12:43:42	35	0.35	34.65

图 13.23　账目核对页面

13.8　结算申请表

医疗机构在与人力资源和社会保障局进行结算以前，需要打印一份《城镇基本医疗保险门诊、药店费用结算申请表》。根据数据汇总方式的不同，打印三种不同格式的结算申请表。在图 13.23 所示的账目核对页面中，单击“汇总打印”、“明细打印”或者“区县汇总”按钮，即分别进入三种不同格式的申请表页面。本节将介绍结算申请表的生成和打印。

13.8.1　汇总表

汇总格式的结算申请表列出了指定医疗机构在指定时间范围内社保卡消费总额和人数总计，表格式如图 13.24 所示。

打印表格　关闭窗口　实际拨付金额　审核确认

[errorMessage]

城镇基本医疗保险门诊、药店费用结算申请表（城镇职工）

Label
Label

人次	费用总额	统筹基金支付	账户消费金额	银联手续费	应拨付金额	实际拨付金额
0	0	0.00	0	0	0	0
根据对帐情况，本期申请划拨资金：　元。						
医疗机构意见： 负责人： （公章） 年　月　日			医保机构意见： 经手人： 科室负责人： 经办机构负责人： （公章） 年　月　日			
备注：本表一式两份，医保经办机构、定点门诊或药店各存一份。 附：城镇基本医疗保险门诊、药店费用结算花名册。						

操作员：Label　　打印时间：

图 13.24　结算申请表（汇总格式）

提示： 本节只介绍结算申请表的生成和打印，不涉及结算申请表的审核和资金结算。图 13.24 中所示页面中审核确认按钮的功能将在本章 18.9 中实现。

（1）在业务实体层项目 Account.Entity 中添加一个类 AccountCheckReport 以描述结算申请表的各个字段。

```
public class CheckAccountReport
{
    public string county { get; set; }          //区县名称
    public int count { get; set; }              //总人数
    public double amt { get; set; }             //消费金额
    public double fee { get; set; }             //手续费
    public double realamt { get; set; }         //实际结算金额
}
```

（2）在数据访问层项目 AccountCheckDAL 中添加一个类 AccountCheckTable，此类主要功能是根据对账时产生的临时表中的数据进行汇总，得到各种报表。

```
//根据对账产生的临时表中数据汇总得到报表数据
```

```
public class AccountCheckTable
{

    private string table;                      //临时表名称
    public AccountCheckTable(string tempTable)
    {
        table = tempTable;
    }
    //生成汇总报表数据
    public CheckAccountReport getSummary()
    {
        return getCountySummary(null);
    }
    //按照参保人员区县进行汇总
    public List<CheckAccountReport> getAllCountySummary()
    {
        //得到临时表中所有不同的区县列表
        string sql = "select distinct brqx from "+table;
       IDataReader reader = MyDbUtility.oracle.ExecuteReader(CommandType.
        Text, sql);
        List<string> counties = new List<string>();
        while (reader.Read())
        {
            if (reader[0] != null)
                counties.Add(reader[0].ToString());
        }
        reader.Close();
        //针对每个区县，得到区县汇总数据
        List<CheckAccountReport> list = new List<CheckAccountReport>();
        foreach (string county in counties)
            list.Add(getCountySummary(county));
        //添加合计数据
        CheckAccountReport report = getSummary();
        report.county = "合计：";
        list.Add(report);
        return list;
    }
    /// <summary>
    ///得到在此医院就医的某区县病人汇总数据
    /// </summary>
    /// <param name="table">临时数据表名</param>
    /// <param name="county">区县名称，为空则所有区县</param>
    private CheckAccountReport getCountySummary(string county)
    {
        bool filter = !string.IsNullOrEmpty(county);   //是否指定了区县
        string where = filter ? (" where brqx='" + county + "'  ") : "";
        CheckAccountReport result = new CheckAccountReport();
        //得到就诊总人次数，如商户号、终端号、卡号都相同，则认为是同一笔交易（一人次）
        string sql = "select count(*) from "
            +" (select distinct corno,term,accno,to char(time,'yyyy-mm-dd
            hh24')"
            +" from "  + table + where + " )";
        result.count = Convert.ToInt32(MyDbUtility.oracle.ExecuteScalar
        (CommandType.Text, sql));
        //得到汇总金额
        sql = "select sum(nvl(amt,0)),sum(nvl(fee,0)) ,sum(nvl(realamt,0))
        from " + table+where;
        var reader = MyDbUtility.oracle.ExecuteReader(CommandType.Text,
        sql);
```

```
        if (reader.Read())
        {
            object o = reader[0];
            result.amt = (o == DBNull.Value ? 0 : Convert.ToDouble(o));
            o = reader[1];
            result.fee = (o == DBNull.Value ? 0 : Convert.ToDouble(o));
            o = reader[2];
            result.realamt = (o == DBNull.Value ? 0 : Convert.ToDouble(o));
        }
        result.county = county;
        return result;
    }
    //得到核对正确的数据汇总
    public DataTable getCorrectSummary()
    {
        string sql = "select name as 商户名称,sum(amt) as 交易金额,sum(fee) as
        手续费,sum(realamt) as 结算金额 from " + table + " where error_flag='0'
        group by name";
        IDataReader reader=MyDbUtility.oracle.ExecuteReader(CommandType.
        Text, sql);
        DataTable t = new DataTable();
        t.Load(reader);
        return t;
    }
    //得到核对错误的数据汇总
    public DataTable getErrorSummary()
    {
        string sql = "select name as 商户名称,sum(amt) as 交易金额,sum(fee) as
        手续费,sum(realamt) as 结算金额 from " + table + " where error flag='1'
        group by name";
       IDataReader reader = MyDbUtility.oracle.ExecuteReader(CommandType.
        Text, sql);
        DataTable t = new DataTable();
        t.Load(reader);
        return t;
    }
    public DataTable getCorrectData(PageDataArgument pageArg)
    {
        return getData( "0",pageArg);
    }
    public DataTable getErrorDate(PageDataArgument pageArg)
    {
        return getData( "1",pageArg);
    }
    /// <summary>
    /// 得到银行数据
    /// </summary>
    /// <param name="errorFlag">错误标志，1 错误,0 正确，*所有数据</param>
    /// <param name="arg">分页参数</param>
    /// <returns>得到的银行数据</returns>
    private DataTable getData(string errorFlag,PageDataArgument arg)
    {
        int min = arg.min;
        int max = arg.max;
        DataTable result = null;
        //查询用的基本 Select 语句
        string sql = "select * from "
            +"( select rownum as 序号, a.name as 商户名称,a.sfzhm as 身份
            证号, "
```

```
            +" b.xm as 姓名,a.accno as 银行账户, a.time as 消费时间,a.amt as 交
            易金额, "
            +" a.fee as 手续费,a.realamt as 结算金额 from "
            + table + " a left outer join emp natl b on a.sfzhm=b.grbh ";
        //如果错误标志不等于*，则根据错误标记进行过滤，生成对应的 Where 条件
        if (errorFlag != "*")
            sql += "where (a.error_flag='" + errorFlag + "') ";
        sql += " order by a.time ";
        sql += ")";
        //如果需要刷新记录总数，则调用 MyDbUtility 工具类的方法得到总数
        if (arg.refreshCount)
            arg.count=MyDbUtility.getCount(sql);
        //添加分页查询条件
        sql += " where 序号 between " + min + " and " + max;
        Database oracle = MyDbUtility.oracle;
        DbCommand command = oracle.GetSqlStringCommand(sql);
        //执行分页查询，得到查询结果并返回
        using (IDataReader reader = oracle.ExecuteReader(command))
        {
            result = new DataTable();
            result.Load(reader);
        }
        return result;
    }
    public DataTable getAllData(PageDataArgument arg)
    {
        return getData( "*", arg);
    }
}
```

（3）在业务逻辑层项目 AccountCheckBLL 中添加一个类 AccountCheckBLL，其主要功能为调用数据访问层相应方法，此处省略其代码。

（4）在表现层项目 AccountCheckWeb 中添加一个页面 PrintCheckResult.aspx，页面设计布局如图 13.24 所示，页面代码如下：

```
<head runat="server">
   <style type="text/css" media="print" >
      .noprint{display:none;}                    /*只显示不打印*/
   </style>
   <style type="text/css" media="all">
   body{font-family:宋体; font-size:12px;}
   table{ font-family:宋体; font-size:12px; border:solid 1px; }
   </style>
   <script src="../js/jquery.js" type="text/javascript"></script>
   <script type="text/javascript">
      $(function() {
         $('#printButton').click(printReport);     //打印按钮 Click 事件
         $('#payMoney').change(checkMoney);        //支付金额 Changed 事件
         $('#audit').click(checkMoney);            //审核按钮 Click 事件
         $('#closeButton').click(function() { window.close(); });
                                                   //关闭按钮 Click 事件
         var v1 = $('#payMoney').val();
         $('#payMoneyLabel').html(v1);
      });    //$(function)
      function printReport() {
         window.print();                           //打印当前页面
      }
```

```
        //检查实际拨付金额不能大于剩余拨付金额
        function checkMoney() {
            var v1 = $('#payMoney').val();
            var pay = parseFloat(v1);
            var v2 = $('#realMoney').html();
            var amt = parseFloat(v2);
            $('#payMoneyLabel').html(v1);
            if (pay > amt) {
                alert('实际拨付金额不能大于应拨付金额! 请修改! ');
                return false;
            }
            return true;
        }
    </script>
    <title> 城镇基本医疗保险费用结算  </title>
</head>
<body>
<form id="form1" runat="server">
   <div class="noprint">
      <input id="printButton" type="button" value="打印表格"  />
      <input type="button" id="closeButton" value="关闭窗口" />
         实际拨付金额
      <asp:TextBox ID="payMoney" runat="server"></asp:TextBox>
      <asp:Button  ID="audit" runat="server" Text="审核确认" onclick=
      "verify Click" /><br />
      <!--错误信息-->
      <asp:Label ID="errorMessage" runat="server" Text="" ForeColor=
      "Red"></asp:Label>
   </div>
   <!--结算申请表表头开始-->
    <span style="font-family: 宋体; font-size: 14px; font-weight: bold;">
    城镇基本医疗保险门诊、药店费用结算申请表(城镇职工)</span><br /><br />
    <asp:Label ID="hospital" runat="server" Text="Label"></asp:Label><br
      />
    <asp:Label ID="dateRange" runat="server" Text="Label"></asp:Label><br
      />
    <!--结算申请表表头结束-->
<!--此 table 为申请表正文-->
    <table rules="all" cellspacing="0" cellpadding="4">
      <tr><td>人次</td><td>费用总额</td>
      <td>统筹基金支付</td><td>账户消费金额</td>
      <td>银联手续费</td><td>应拨付金额</td>
      <td>实际拨付金额</td></tr>
      <tr>
         <td><asp:Label ID="transactionCount" runat="server" Text="0">
         </asp:Label></td>
         <td><asp:Label ID="totalMoney" runat="server" Text="0">
         </asp:Label></td>
         <td>0.00</td><td>
         <asp:Label ID="totalMoney2" runat="server" Text="0">
         </asp:Label></td>
         <td><asp:Label ID="totalFee" runat="server" Text="0">
         </asp:Label></td>
         <td><asp:Label ID="realMoney" runat="server" Text="0">
         </asp:Label></td>
         <td><asp:Label ID="payMoneyLabel" runat="server" Text="0">
         </asp:Label></td>
         </tr>
```

```
      <tr><td colspan="7">根据对账情况，本期申请划拨资金：
      <asp:Label runat="server" id="applyMoney" Width="150"></asp:Label>
      元。</td>
      </tr>
      <tr>
      <td colspan="4">
      <pre> 医疗机构意见：



      负责人：
                  （公章）
                 年    月    日
      </pre></td>
      <td colspan="3">
      <pre> 医保机构意见：


        经手人：
    科室负责人：
  经办机构负责人：
                  （公章）
                 年    月    日
      </pre>
      </td>
      </tr>
      <tr><td colspan="6">
      备注：本表一式两份，医保经办机构、定点门诊或药店各存一份。<br />
      附:城镇基本医疗保险门诊、药店费用结算花名册。     <br />
      </td></tr>
     </table>
     操作员：<asp:Label ID="userName" runat="server" Text="Label">
     </asp:Label>
         打印时间：<%=DateTime.Today.ToShortDateString()%>
   </form>
</body>
```

（5）在 PrintCheckResult.aspx 页面的 Page_Load 事件中，得到报表数据并显示在相应控件上。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        //设计审核按钮的可见性，对医疗机构不可见
        audit.Visible = !Common.WebUtility.isMedicineShopUser();
        generateReport();                              //生成报表数据
    }
}
//生成报表数据
private void generateReport()
{
    //如果输入日期不合法则返回
    if (!checkDateArg())
        return;
    CheckAccountArg arg = Session[Common.Constants.SessionCheckArg] as
    CheckAccountArg;                                   //得到对账参数
```

```
    Hospital h = HospitalBLL.getByID(arg.hospital, arg.institute);
    hospital.Text = "医疗机构编码：" + arg.hospital + "  医疗机构
    名称：" + h.Name;
    AccountCheckBLL bll = new AccountCheckBLL(Common.WebUtility.
    currentUser.id);
    CheckAccountReport report = bll.getSummary();      //得到汇总数据
    //根据汇总数据设置各个控件的值
    totalMoney.Text = totalMoney2.Text=string.Format("{0:0.00}",
    report.amt);
    totalFee.Text = string.Format("{0:0.00}", report.fee);
    realMoney.Text = string.Format("{0:0.00}", report.realamt);
    payMoney.Text = string.Format("{0:0.00}", report.realamt);
    transactionCount.Text = report.count.ToString();
    userName.Text = Common.WebUtility.currentUser.name;
}
//检查并显示日期范围
private bool checkDateArg()
{
    //从 Session 中获得对账检索参数
    CheckAccountArg arg = Session[Common.Constants.SessionCheckArg] as
    CheckAccountArg;
    if (arg == null)
    {
        errorMessage.Text = "未能正确设置对账参数，不能显示和打印报表。";
        return false;
    }
    dateRange.Text = "起始日期："+arg.beginDate.ToShortDateString()
        +"  终止日期："+arg.endDate.ToShortDateString();
    return true;
}
```

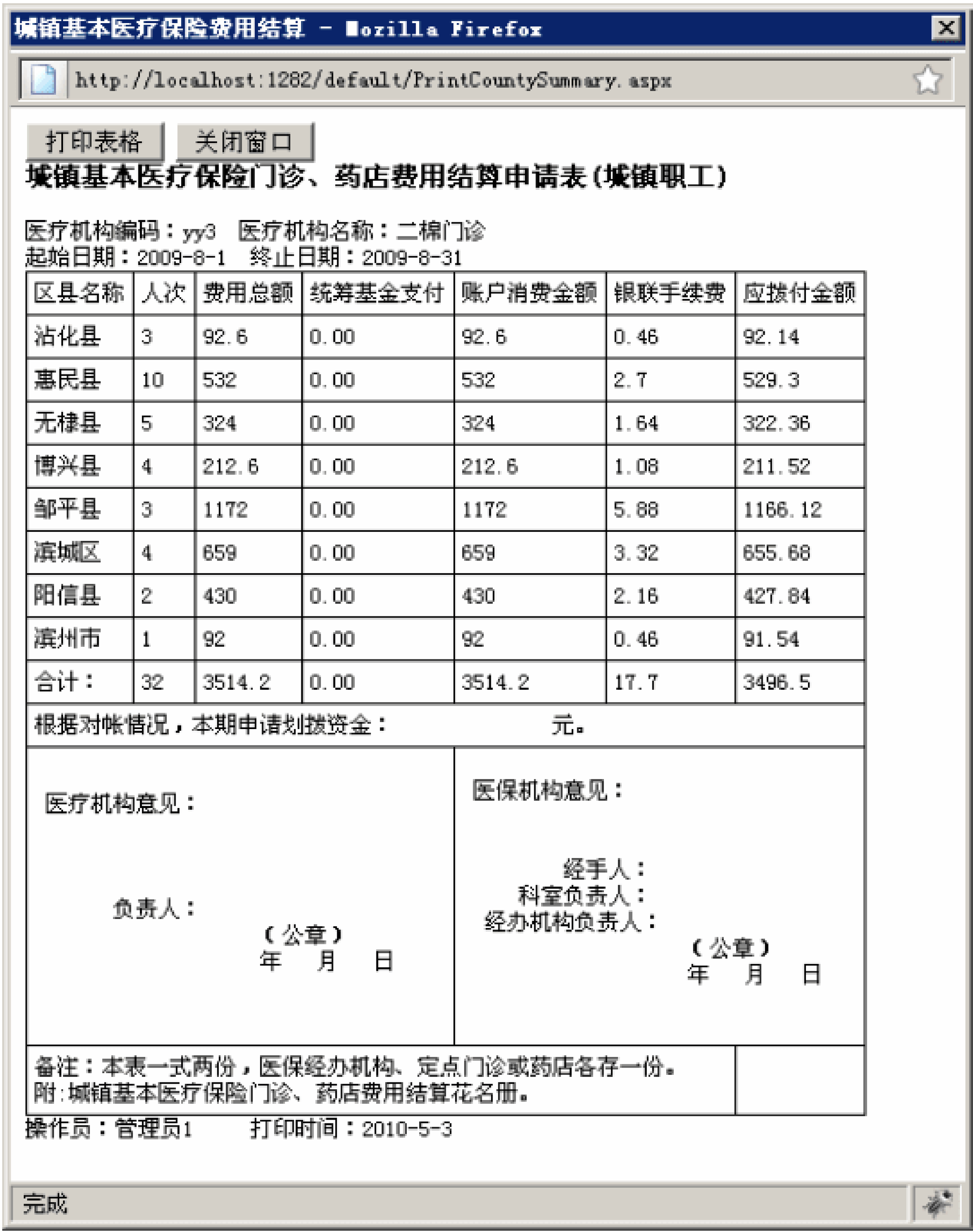
（6）运行 PrintCheckResult.aspx 页面，运行结果如图 13.25 所示。

图 13.25　资金结算申请单（汇总格式）运行页面

13.8.2　区县汇总表

结算申请表的第二种格式为区县汇总表，可以统计某医疗机构就诊的人员来自于哪个

区县，以区县为单位进行汇总统计。区县汇总格式的结算申请运行界面如图 13.26 所示。



城镇基本医疗保险门诊、药店费用结算申请表(城镇职工)

医疗机构编码：yy3　医疗机构名称：二棉门诊
起始日期：2009-8-1　终止日期：2009-8-31

区县名称	人次	费用总额	统筹基金支付	账户消费金额	银联手续费	应拨付金额
沾化县	3	92.6	0.00	92.6	0.46	92.14
惠民县	10	532	0.00	532	2.7	529.3
无棣县	5	324	0.00	324	1.64	322.36
博兴县	4	212.6	0.00	212.6	1.08	211.52
邹平县	3	1172	0.00	1172	5.88	1166.12
滨城区	4	659	0.00	659	3.32	655.68
阳信县	2	430	0.00	430	2.16	427.84
滨州市	1	92	0.00	92	0.46	91.54
合计：	32	3514.2	0.00	3514.2	17.7	3496.5

根据对帐情况，本期申请划拨资金：　　　元。

医疗机构意见：

负责人：
（公章）
年　月　日

医保机构意见：

经手人：
科室负责人：
经办机构负责人：
（公章）
年　月　日

备注：本表一式两份，医保经办机构、定点门诊或药店各存一份。
附：城镇基本医疗保险门诊、药店费用结算花名册。

操作员：管理员1　　打印时间：2010-5-3

图 13.26　结算申请表（区县汇总格式）

（1）在表现层项目 AccountCheckWeb 中添加一个页面 PrintCountySummary.aspx。此页面代码与汇总格式的结算申请表大致相同，只是表头下面用一个 Repeater 控件循环显示了所有区县的汇总数据，如下代码所示。

```
<table rules="all" cellspacing="0" cellpadding="4">
<tr><td>区县名称</td><td>人次</td><td>费用总额</td><td>统筹基金支付</td>
<td>账户消费金额</td><td>银联手续费</td><td>应拨付金额</td>
</tr>
<asp:Repeater ID="Repeater1" runat="server">
<ItemTemplate>
<tr><td><%#Eval("county") %></td>
<td><%#Eval("count") %></td>
<td><%#Eval("amt") %></td>
<td>0.00</td>
<td><%#Eval("amt")%></td>
<td><%#Eval("fee")%></td>
<td><%#Eval("realamt")%></td>
</tr>
</ItemTemplate>
</asp:Repeater>
```

（2）在数据访问层项目 AccountCheckDAL 中添加一个类 SbjgDB，以更新和获取社保机构所属区县。

```
public static class SbjgDB
{
    //更新社保机构所属区县
    public static void updateCounty()
    {
```

```
        //表说明：si.si_natl 为社保机构总表,md.sbjgqx 为社保机构所属区县表
        //将 si_natl 中存在而 sbjgqx 表中不存在的社保机构添加到 sbjgqx 中
        string sql = " insert into md.sbjgqx (sbjgbh, sbjgmc)"
            +"  select sbjgbh, sbjgmc from si.si_natl "
            + " where sbjgbh not in (select sbjgbh from md.sbjgqx)";
        Database db = MyDbUtility.oracle;
        int n=db.ExecuteNonQuery(CommandType.Text, sql);
        if (n == 0) return;
        //查询 sbjgqx 中所属区县为空的社保机构
        sql = "select sbjgbh,sbjgmc,ssqx from md.sbjgqx where ssqx is null";
        DataTable table = db.ExecuteDataSet(CommandType.Text, sql).
        Tables[0];
        string sbjgbh, sbjgmc, qx;
        //生成 update 语句
        sql = "update md.sbjgqx set ssqx=:qx where sbjgbh=:bh";
        DbCommand command = db.GetSqlStringCommand(sql);
        db.AddInParameter(command, ":qx", DbType.String);
        db.AddInParameter(command, ":bh", DbType.String);
        foreach (DataRow row in table.Rows)    //循环更新新机构所属区县
        {
            sbjgbh = row[0].ToString();
            sbjgmc = Convert.ToString(row[1]);
            qx = getCounty(sbjgmc);
            command.Parameters[":qx"].Value = qx;
            command.Parameters[":bh"].Value = sbjgbh;
            db.ExecuteNonQuery(command);
        }
    }
    //此数组包含了 A 市所有区县
    static string[] couties={"滨城区","博兴县","惠民县","无棣县","阳信县",
                              "沾化县","邹平县","经济开发区","高新区"};
    //根据社保机构名称判断社保机构所属区县
    public static string getCounty(string sbjgmc)
    {
        //数组中已有区县直接返回
        foreach(string c in couties)
            if(sbjgmc.IndexOf(c)>=0)
                return c;
        int qx;
        //在社保机构名称中查找"区"或者"县"出现的位置
        if ((qx=sbjgmc.IndexOf('县')) > 0 || (qx=sbjgmc.IndexOf('区')) > 0)
        {
            //"市"之后"县"之前的文本即为区县名称，返回这个字符串
            int city=sbjgmc.IndexOf('市');
            if (city < 0)
                return sbjgmc.Substring(0, qx + 1);
            return sbjgmc.Substring(city+1, qx - city );
        }
        //如果社保机构不包含县或区字样，则查找"A 市"字样
        if (sbjgmc.IndexOf("A 市") >= 0)
            return "A 市";
        throw new ApplicationException("不能判断这个社保机构所属区县:
        "+sbjgmc);
    }
    //得到所有社保机构编号
    public static List<string> getAll()
    {
        string sql = "select distinct sbjgbh  from md.institution_natl";
```

```
        IDataReader reader= MyDbUtility.oracle.ExecuteReader
        (CommandType.Text, sql);
        List<string> list = new List<string>();
        while (reader.Read())
        {
            list.Add(reader[0].ToString());
        }
        return list;
    }
}
```

（3）在 PrintCountySummary.aspx 页面的 Page_Load 事件中，生成并显示报表数据。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        SbjgBLL.updateCounty();        //更新社保机构所属区县
        generateReport();              //生成报表
    }
}
private void generateReport()
{
    if (!checkDateArg())
        return;
    CheckAccountArg arg = Session[Common.Constants.SessionCheckArg] as
    CheckAccountArg;                   //得到对账参数
    //取得医院信息并显示在页面上
    Hospital h = HospitalBLL.getByID(arg.hospital, arg.institute);
    hospital.Text = "医疗机构编码：" + arg.hospital + "  医疗机构
    名称：" + h.Name;
    //生成报表汇总数据
    AccountCheckBLL bll = new AccountCheckBLL(Common.WebUtility.
    currentUser.id);
    List<CheckAccountReport> report = bll.getAllCountySummary();
    //将数据绑定到 Repeater 控件
    Repeater1.DataSource = report;
    Repeater1.DataBind();
    userName.Text = Common.WebUtility.currentUser.name;
}
```

13.9　审核和结算

医疗机构提出结算申请以后，人力资源和社会保障局工作人员对这些数据进行审核，并进行结算支付。资金可以一次结清，也可以分成若干次结清。

13.9.1　实体类设计

根据用户需求，在审核医疗机构的结算申请时，需要保存相关审核信息，如审核人、审核日期、各种金额等。医疗机构的结算申请通过审核以后，人力资源和社会保障局就会对此次审核费用进行结算。资金可以一次结清，也可以分成若干次结清。由此可以看出，

审核和结算之间是一对多的关系，一次审核可以包含一到多次结算，而一次结算必然只属于一次审核。与此相关的实体类有两个：审核信息类 AuditInfo 和审核支付类 AuditPayment。两个类的代码如下：

```
public class AuditInfo
{
    public string auditNo{get;set;}              //审核编号
    public DateTime auditDate{get;set;}          //审核日期
    public string auditUser{get;set;}            //审核人
    public string yybm{get;set;}                 //医疗编码
    public string sbjgbh{get;set;}               //社保机构编号
    public string yymc{get;set;}                 //医院名称
    public double amt{get;set;}                  //消费金额
    public double fee{get;set;}                  //手续费
    public double realamt{get;set;}              //实际结算金额
    public double payMoney { get; set; }         //已经支付金额
    public string cancel { get; set; }           //撤销标志(审核是否被撤销)
    //剩余金额（此交审核尚未结算的金额）
    public double moneyRemained
    {
        get { return realamt - payMoney; }
    }
}
public class AuditPayment
{
    public string auditNo { get; set; }          //审核编号
    public int payNo { get; set; }               //结算支付顺序号
    public DateTime payDate { get; set; }        //支付日期
    public double payMoney { get; set; }         //支付金额
    public string payUser { get; set; }          //支付用户
    public string cancel { get; set; }           //撤销标志
    public const int NewPayNo = -1;              //新增的资金支付序号
    public AuditPayment()
    {
        payNo = NewPayNo;
    }
}
```

13.9.2　数据访问层和业务逻辑层

通过与用户交流，得知审核和结算是密切相关的两个操作。通常医疗机构提出某个时间段的结算申请后，人力资源和社会保障局工作人员审核这个时间段的费用，并且进行资金结算。数据访问层和业务逻辑层的主要功能是添加及查询审核和支付数据，根据用户需求，审核和支付数据不能被修改和删除，但是审核可以被撤销。

如前所述，审核和结算支付之间是一对多的关系，一次审核可以包含一到多次支付，而一次支付必然只属于一次审核。为了标识同一次审核所包含的多次支付之间的顺序关系，第一次支付都被分配一个顺序号，第一次支付的顺序号为 1，第二次为 2，依次类推。

（1）在数据访问层项目 AccountCheckDAL 中添加一个类 AuditPayDB，这个类实现了与支付相关的数据访问功能。代码如下：

```
//结算支付数据访问类
public static class AuditPayDB
{
    //数据库表名和表中各个字段名称
    static string table = " Audit Payment ";
   static string columns = " Audit No,Pay No,Pay Date,Pay Money,Pay User,
    Cancel ";
    /// <summary>
    /// 根据审核编号得到审核信息
    /// </summary>
    /// <param name="audit">审核编号</param>
    public static List<AuditPayment> getByAudit(string audit)
    {
        string sql = " select " + columns +" from "+ table
            + " where Audit No=:audit0 order by pay no";
        var oracle = MyDbUtility.oracle;
        var command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":audit0", DbType.String, audit);
        var reader = oracle.ExecuteReader(command);
        var list = fromReader(reader);
        reader.Close();
        return list;
    }
    /// <summary>
    /// 添加支付信息
    /// </summary>
    /// <param name="pay">要添加的支付信息</param>
    /// <remarks>
    /// 此方法只应从 AuditDB.add()方法中调用（当增加新的支付时），
    /// 不应该从页面中或其他代码中调用，以免产生数据不一致。
    /// </remarks>
    internal static int add(AuditPayment pay)
    {
        string sql = "insert into " + table + " ( " + columns
            + ") values (:audit0 no,:pay no,sysdate,:money0,:user0,'0')";
        if (pay.payNo == AuditPayment.NewPayNo)
            pay.payNo = newPayNo(pay.auditNo);
        var oracle = MyDbUtility.oracle;
        var command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":audit0 no", DbType.String, pay.
        auditNo);
        oracle.AddInParameter(command, ":pay_no", DbType.Int32, pay.
        payNo);
        oracle.AddInParameter(command, ":money0", DbType.Double, pay.
        payMoney);
        oracle.AddInParameter(command, ":user0", DbType.String, pay.
        payUser);
        return oracle.ExecuteNonQuery(command);
    }
    /// <summary>
    /// 得到新的支付顺序号
    /// </summary>
    /// <param name="audit">审核编号</param>
    /// <returns>新得到的支付顺序号</returns>
    /// <remarks>
    /// 一次审核可以对应多次支付，第一次支付序号为 1，第二次为 2，依次类推
    /// </remarks>
    private static int newPayNo(string audit)
    {
```

```
        string sql = " select nvl(max(Pay_No),0) from Audit_Payment where
        Audit_No=:audit0";
        var oracle = MyDbUtility.oracle;
        var command = oracle.GetSqlStringCommand(sql);
        oracle.AddInParameter(command, ":audit0", DbType.String, audit);
        int n = Convert.ToInt32(oracle.ExecuteScalar(command));
        return n+1;
    }
    //从 DataReade 循环读取数据数据生成支付信息列表
    private static List<AuditPayment> fromReader(IDataReader reader)
    {
        List<AuditPayment> list = new List<AuditPayment>();
        while (reader.Read())
        {
            AuditPayment p = new AuditPayment();
            p.auditNo = Convert.ToString(reader[0]);
            p.payNo = Convert.ToInt32(reader[1]);
            p.payDate = Convert.ToDateTime(reader[2]);
            p.payMoney = Convert.ToDouble(reader[3]);
            p.payUser = Convert.ToString(reader[4]);
            p.cancel = Convert.ToString(reader["cancel"]);
            list.Add(p);
        }
        return list;
    }
}
```

（2）在数据访问层项目 AccountCheckDAL 中添加一个类 AuditDB，这个类实现了与审核相关的数据访问功能。代码如下：

```
public static class AuditDB
{
    #region 数据库表名和字段名称
    private const string AuditTableName = "md.bank_transaction_audit";
    private const string AuditNoColumn = "audit_no";
    private const string AuditDateColumn = "audit_date";
    private const string AuditUserColumn = "audit_user";
    private const string HospitalIDColumn = "yybm";
    private const string InsititeIDColumn = "sbjgbh";
    private const string HospitalNameColumn = "yymc";
    private const string AmtSumColumn = "amt_sum";
    private const string FeeSumColumn = "fee_sum";
    private const string RealAmtSumColumn = "realamt_sum";
    private const string PayMoneyColumn = "money_payed";
    private const string CancelColumn = "cancel";
    #endregion
    //添加审核记录
    public static int audit(AuditInfo audit)
    {
    ...                            //参考光盘源代码
    }
    /// <summary>
    /// 支付资金
    /// </summary>
    public static int payMoney(AuditPayment payment)
    {
    ...                            //参考光盘源代码
    }
    /// <summary>
    /// 撤销审核
```

```
    /// </summary>
    /// <param name="audit">要撤销的审核 ID</param>
    /// <param name="user">用户 ID</param>
    public static int cancel(string audit,string user)
    {
    ...                                //参考光盘源代码
    }
    /// <summary>
    /// 根据条件得到审核列表
    /// </summary>
    /// <param name="sbjgbh">社保机构编号</param>
    /// <param name="dateArg">日期范围</param>
    /// <param name="pageArg">分页参数</param>
    public static List<AuditInfo> getAuditList(string sbjgbh,DateRangeArg
    dateArg, PageDataArgument pageArg)
    {
    ...                                //参考光盘源代码
    }
    /// <summary>
    /// 根据审核编号得到审核信息
    /// </summary>
    public static AuditInfo getByID(string audit)
    {
    ...                                //参考光盘源代码
    }
    /// <summary>
    /// 从 DataReader 中循环读取数据并创建审核信息列表
    /// </summary>
    private static List<AuditInfo> fromReader(IDataReader reader)
    {
    ...                                //参考光盘源代码
    }
    /// <summary>
    /// 得到新的审核编号（格式为当前年月日 6 位+4 位顺序号，共 10 位字符）
    /// </summary>
    /// <returns></returns>
    private static string getNewID()
    {
    ...                                //参考光盘源代码
    }
}
```

（3）业务逻辑层的主要功能是调用数据访问层的相应方法，此处省略详细代码。

13.9.3　审核结算页面

在图 13.25 所示的结算申请表（汇总格式）页面中，用户可以单击输入一定结算金额，然后单击“审核确认”按钮完成审核和结算支付功能，代码如下：

```
protected void verify Click(object sender, EventArgs e)
{
    //生成审核对象并设置各个属性
    AuditInfo audit2 = new AuditInfo();
    WebUser user = Common.WebUtility.currentUser;
    audit2.amt = double.Parse(totalMoney.Text);
    audit2.auditUser = user.id;
```

```
    audit2.fee = double.Parse(totalFee.Text);
    CheckAccountArg arg = Session[Common.Constants.SessionCheckArg] as
    CheckAccountArg;
    Hospital h = HospitalBLL.getByID(arg.hospital, arg.institute);
    audit2.yybm=arg.hospital;
    audit2.sbjgbh = arg.institute;
    audit2.yymc = h.Name;
    audit2.realamt = double.Parse(realMoney.Text);
    audit2.payMoney = double.Parse(payMoney.Text);
    AuditBLL.audit(audit2);              //添加审核信息
    audit.Enabled = false;
    audit.Visible = false;
    Page.ClientScript.RegisterStartupScript
        (this.GetType(),"auditok", "<script>alert('审核成功！数据已被记录！
        ');</script>");
}
```

13.9.4　二次结算页面

如前所述，对医疗机构的结算可以一次性完成，也可以分成多次支付。除第一次结算支付外，其余结算支付均称为二次结算。二次结算时，用户首先查询某次审核对应的结算情况，如已支付金额和余额，然后输入一定的金额（不大于余额）进行结算。二次结算页面设计外观如图 13.27 所示。

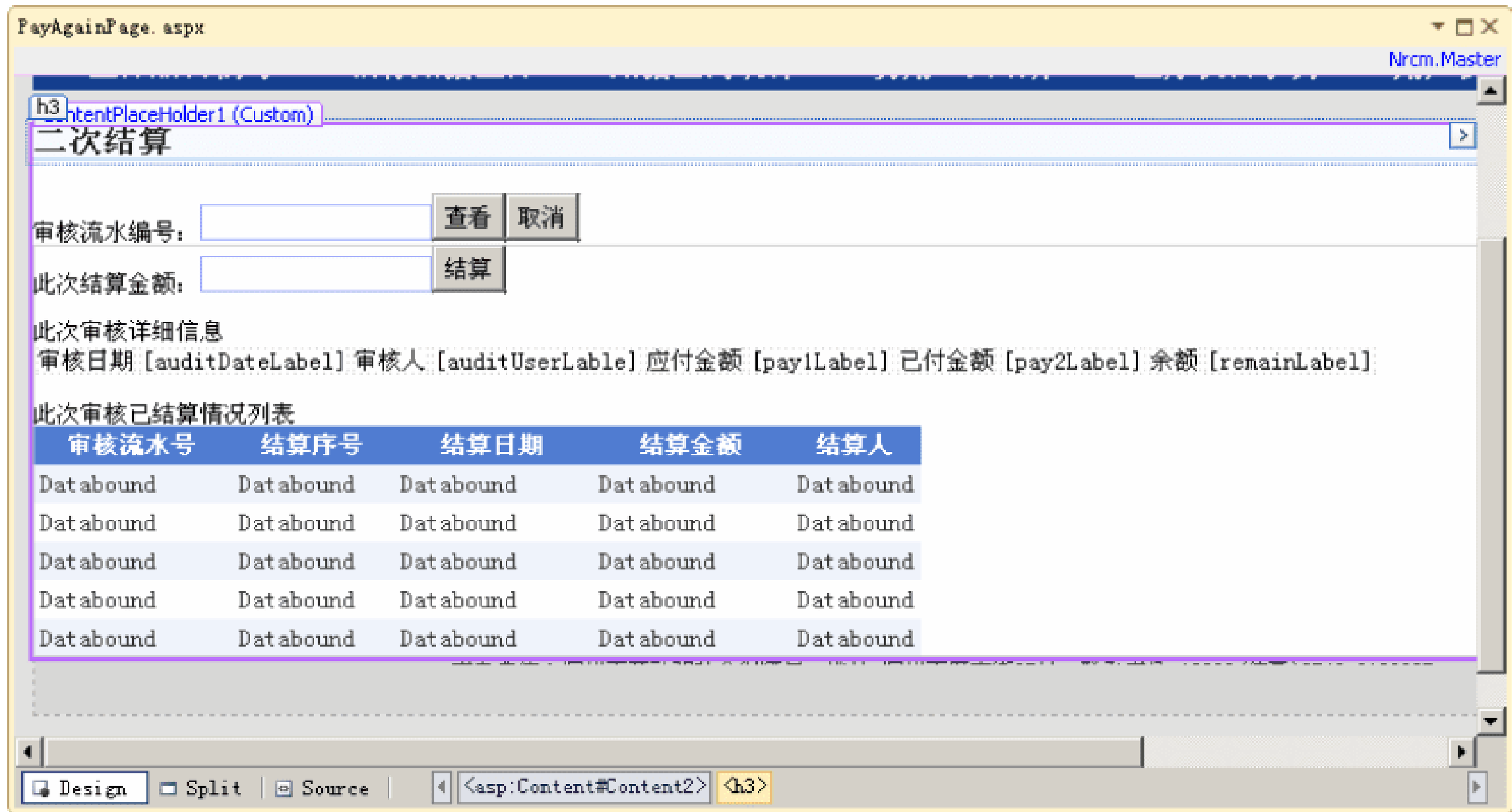


图 13.27　二次结算页面设计外观

（1）在表现层项目 AccountCheckWeb 中添加一个用户控件 PaymentListControl.ascx，此控件显示了某次审核的所有支付列表。控件代码如下：

```
//PaymentListControl.ascx 文件代码
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="PaymentList
Control.ascx.cs"
Inherits="AccountCheckWeb.UserControls.PaymentListControl" %>
<asp:GridView runat="server" ID="grid1" CssClass="gridview" AutoGenerate
Columns="false">
<Columns>
<asp:BoundField DataField="AuditNo" HeaderText="审核流水号"
ItemStyle-Width="100" />
```

```
<asp:BoundField DataField="PayNo" HeaderText="结算序号" ItemStyle-
Width="80" />
<asp:BoundField DataField="PayDate" HeaderText="结算日期"
DataFormatString="{0:D}" ItemStyle-Width="100" />
<asp:BoundField DataField="PayMoney" HeaderText="结算金额"
DataFormatString="{0:C}" ItemStyle-Width="100" />
<asp:BoundField DataField="PayUser" HeaderText="结算人" />
</Columns>
<EmptyDataTemplate>
<span style="color:Red;"> 没有数据。</span>
</EmptyDataTemplate>
</asp:GridView>
//PaymentListControl.ascx.cs 文件中的 C#代码
public partial class PaymentListControl : System.Web.UI.UserControl
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (needReload)
        {
            loadData();
        }
    }
    //加载支付数据
    private void loadData()
    {
        if (!needReload) return;
        List<Account.Entity.AuditPayment> list = AuditPayBLL.getByAudit
        (auditNo);
        grid1.DataSource = list;
        grid1.DataBind();
        needReload = false;
    }
    public void refresh()
    {
        needReload = true;
    }
    //私有属性：是否需要重新加载数据
    private bool needReload
    {
        get
        {
            string s = ViewState["needReload"] as string;
            return s == "true";
        }
        set
        {
            if (value)
                ViewState["needReload"] = "true";
            else
                ViewState["needReload"] = "false";
        }
    }
    //审核编号
    public string auditNo
    {
        get
        {
            return ViewState["theAuditNo"] as string;
        }
        set
```

```
        {
            string old = ViewState["theAuditNo"] as string;
            if (old == value) return;
            ViewState["theAuditNo"] = value;
            needReload = true;          //审核编号改变后，数据需要重新加载
            loadData();
        }
    }
}
```

（2）在表现层项目 AccountCheckWeb 中添加一个页面 PayAgainPage.aspx，页面代码如下：

```
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
   <script type="text/javascript">
       //检测支付金额是否合理
      function checkMoney() {
         var remain = $('#<%=remainLabel.ClientID%>').text();
         var pay = $('#<%=payMoney.ClientID%>').val();
          if(parseFloat(pay) <= 0)
          {
              alert('支付金额必须大于零');
              return false;
          }
         if (parseFloat(pay) > parseFloat(remain)) {
            alert('此次支付金额不能大于余额！请修改！');
            return false;
         }
         return true;
      }
   </script>
</asp:Content>
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
   <h3>二次结算</h3>
   审核流水编号：<asp:TextBox ID="auditNoText" runat="server" />
   <asp:Button ID="ok"  runat="server" Text="查看" onclick="ok_Click" />
   <asp:Button ID="cancel" runat="server" Text="取消" onclick=
   "cancel_Click" />
   <asp:Panel runat="server" ID="payPanel">
      此次结算金额：<asp:TextBox runat="server" ID="payMoney" />
      <asp:Button runat="server" ID="payButton" Text="结算" OnClientClick=
      "return checkMoney();" onclick="payButton_Click" /><br />
      <br />此次审核详细信息<br />
      <table>  <tr>
      <td>审核日期</td><td><asp:Label runat="server" ID="auditDateLabel"
      /></td>
      <td>审核人</td><td><asp:Label runat="server" ID="auditUserLable"
      /></td>
      <td>应付金额</td><td><asp:Label runat="server" ID="pay1Label"
      /></td>
      <td>已付金额</td><td><asp:Label runat="server" ID="pay2Label"
      /></td>
      <td>余额</td><td><asp:Label runat="server" ID="remainLabel" /></td>
      </tr></table>
```

```
        <br />
        此次审核已结算情况列表<br />
        <uc1:PaymentListControl ID="payList1" runat="server"/>
    </asp:Panel>
</asp:Content>
```

（3）在 PayAgainPage.aspx 页面中“查看”按钮的 Click 事件中，根据用户输入的审核编号取出审核信息及支付列表，并显示在页面上。

```
protected void ok Click(object sender, EventArgs e)
{
    payMode(false);
    AuditInfo audit=AuditBLL.getByID(auditNoText.Text);     //查找审核信息
    if (audit == null) return;
    if (audit.moneyRemained <= 0)
    {
        ClientScript.RegisterStartupScript(this.GetType(),"NoRemain",
            "<script>alert('此次审核已经结算完毕，不能再次结算！');</script>");
        return;
    }
    payMode(true);                                          //进入支付模式
    //将得到的审核信息显示在各个控件中
    auditUserLable.Text = audit.auditUser;
    auditDateLabel.Text = audit.auditDate.ToShortDateString();
    pay1Label.Text = audit.realamt.ToString("0.00");
    pay2Label.Text = audit.payMoney.ToString("0.00");
    remainLabel.Text = audit.moneyRemained.ToString("0.00");
    payList1.auditNo = auditNoText.Text;
}
//是否进入支付模式（不同模式下各个控件可见性、可用性不同）
private void payMode(bool flag)
{
    payPanel.Visible = flag;
    payMoney.ReadOnly = !flag;
    payButton.Enabled = flag;
    auditNoText.ReadOnly = flag;
}
```

（4）在 PayAgainPage.aspx 页面“支付”按钮的 Click 事件中，完成支付功能。

```
protected void payButton Click(object sender, EventArgs e)
{
    AuditPayment pay = new AuditPayment();
    pay.auditNo = auditNoText.Text;
    pay.payMoney = double.Parse(payMoney.Text);
    pay.payUser = WebUtility.currentUser.id;
    AuditBLL.payMoney(pay);
    ClientScript.RegisterStartupScript(this.GetType(),
        "successpay", "<script>alert('结算成功！');</script>");
    payMode(false);
    payList1.refresh();                                     //刷新支付列表
}
```

（5）运行 PayAgainPage.aspx 页面，运行界面如图 13.28 所示。



图 13.28　二次结算页面运行界面

13.10　统 计 报 表

根据用户需求，用户可以对社保卡结算数据进行统计生成报表。用户要求产生 5 种统计报表，由于有些报表实现方式类似，受篇幅限制，本节只介绍其中两种报表的设计和实现。

13.10.1　审核结算明细表

在审核结算明细表中，用户根据时间范围查询审核结算明细，并可以撤销审核。审核结算明细表界面如图 13.29 所示。



图 13.29　审核结算明细表

（1）在表现层项目 AccountCheckWeb 中添加一个嵌套母版页 ReportMaster.master，此母版页在 Nrcm.master 母版页的基础上添加了左侧导航栏。此母版页是所有报表页面的母版。ReportMaster.master 页面代码如下：

```
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
 <div id="leftNav">
<a href="ConsumeReportPage.aspx">消费汇总统计</a><br /><hr />
<a href="CompensationReportPage.aspx">结算汇总统计</a><br /><hr />
<a href="CompensationStatePage.aspx">结算情况查询</a><br /><hr />
<a href="AuditCompensationPage.aspx">审核情况查询</a><br /><hr />
<a href="PayReportPage.aspx">分批结算查询</a><br /><hr />
</div>
<div id="rightContent">
<asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
</asp:ContentPlaceHolder>
</div>
</asp:Content>
```

（2）在表现层项目 AccountCheckWeb 中，以 ReportMaster.master 为母版页添加一个新页面 AuditCompensationPage.aspx，页面布局如图 13.29 所示，页面代码如下：

```
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
   <h3>审核结算明细表</h3> 社保机构：
   <asp:DropDownList ID="hospitalList"  runat="server"></asp:
   DropDownList>
   日期：<asp:TextBox ID="date1" runat="server"></asp:TextBox>
   至<asp:TextBox  ID="date2" runat="server"></asp:TextBox>  
   <asp:Button ID="OK" runat="server" Text="查询" onclick="OK_Click" /> <br
   />
<div>
    <asp:GridView ID="grid1" runat="server" CssClass="gridview"
        AutoGenerateColumns="false"
        onselectedindexchanging="grid1_SelectedIndexChanging" Data
        KeyNames="AuditNo">
    <Columns>
    <asp:BoundField DataField="AuditNo" HeaderText="流水号" />
    <asp:BoundField DataField="AuditDate" HeaderText="审核日期" Data
    FormatString="{0:yyyy-MM-dd}" />
    <asp:BoundField DataField="AuditUser" HeaderText="审核人" />
    <asp:BoundField DataField="yybm" HeaderText="医院编码" />
    <asp:BoundField DataField="yymc" HeaderText="医院名称" />
    <asp:BoundField DataField="amt" HeaderText="消费金额"
    DataFormatString="{0:0.00}" />
    <asp:BoundField DataField="fee" HeaderText="手续费"  DataFormatString=
    "{0:0.00}"/>
    <asp:BoundField DataField="realamt" HeaderText="应结算金额"
    DataFormatString="{0:0.00}"/>
    <asp:BoundField DataField="payMoney" HeaderText="已结算金额"
    DataFormatString="{0:0.00}"/>
    <asp:TemplateField HeaderText="撤销标志">
    <ItemTemplate>
    <asp:CheckBox runat="server"  ID="cancelFlag"
        Checked=' <%#Eval("cancel").ToString().Trim()=="1"%>' Enabled=
        "false" />
```

```
    </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="详情">
    <ItemTemplate>
    <a href="../report/PayReportPage.aspx?audit=<%#Eval("AuditNo")%>">
    <img src="../images/view.png" alt="查看" style="border:none;" /></a>
    </ItemTemplate>
    </asp:TemplateField>
   <asp:CommandField ShowSelectButton="true" SelectImageUrl=
   "../images/undo.png" ButtonType="Image" HeaderText="撤销" />
    </Columns>
       <EmptyDataTemplate>
          <span style="color:Red; font-weight:bold;">没有符合条件的数据
          </span>
       </EmptyDataTemplate>
    </asp:GridView>
    <webdiyer:AspNetPager ID="pager1" runat="server">
    </webdiyer:AspNetPager>
</div>
</asp:Content>
```

（3）在 AuditCompensationPage.aspx 页面的 Page_Load 事件中，绑定所有社保机构列表以供用户选择。

```
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        bindSbjgList();
    }
}
//绑定社保机构列表
private void bindSbjgList()
{
    List<string> list = SbjgBLL.getAll();
    foreach (string item in list)
    {
        hospitalList.Items.Add(item);
    }
}
```

（4）在 AuditCompensationPage.aspx 页面“查询”按钮的 Click 事件中，根据用户指定的条件查询数据并显示。

```
//绑定审核结算列表
private void bindList()
{
    //创建分页参数
    PageDataArgument page = new PageDataArgument() { refreshCount = true };
    page.pageIndex = pager1.CurrentPageIndex - 1;
    page.pageSize = pager1.PageSize;
    //得到审核数据列表，并绑定到 GridView
    List<AuditInfo> list = AuditBLL.getAuditList(hospitalList.
    SelectedValue,
        DateRangeArg.between2Date(date1.Text, date2.Text),page);
    pager1.RecordCount = page.count;
    grid1.DataSource = list;
    grid1.DataBind();
}
```

```
//单击查询按钮时，根据查询条件显示第 1 页数据
protected void OK_Click(object sender, EventArgs e)
{
    pager1.CurrentPageIndex = 1;
    bindList();
}
```

（5）在 AuditCompensationPage.aspx 页面 GridView 的 SelectedIndexChanged 事件中，撤销 GridView 当前选择的审核。

```
protected void grid1_SelectedIndexChanging(object sender,
GridViewSelectEventArgs e)
{
    e.Cancel = true;                //取消当前事件
    //得到 CheckBox 控件
    CheckBox check = grid1.Rows[e.NewSelectedIndex].FindControl
    ("cancelFlag") as CheckBox;
    //如果当前选中记录已经被撤销则返回
    if (check.Checked)
        return;
    //得到审核编号并撤销此次审核
    string audit = grid1.DataKeys[e.NewSelectedIndex][0].ToString();
    AuditBLL.cancel(audit, AccountCheckWeb.Common.WebUtility.
    currentUser.id);
    WebUtility.registerStartupScript(this, "alert('已经撤销此次审核！');");
    bindList();
}
```

13.10.2　结算情况统计表

在结算情况统计表中，可以查询某社保机构已经结算和尚未结算的社保消费数据，运行界面如图 13.30 所示。



图 13.30　结算情况统计表

（1）在数据访问层项目 AccountCheckDAL 中添加一个类 ReportDB，用以产生各种报表数据。代码如下：

```
public static class ReportDB
```

```
{
    //按照医疗机构进行消费汇总统计
    public static List<HospitalReport> consumeReport(string sbjgbh,
    DateRangeArg date)
    {
        return consumeReportByAudit(sbjgbh, date, '*');
    }
    //按钮医疗机构进行消费汇总（只统计未审核数据）
    public static List<HospitalReport> unauditReport(string sbjgbh,
    DateRangeArg date)
    {
        return consumeReportByAudit(sbjgbh, date, '0');
    }
    /// <summary>
    /// 根据审核结算状态查询消费汇总
    /// </summary>
    /// <param name="sbjgbh">社保机构编号</param>
    /// <param name="date">日期范围</param>
    /// <param name="auditState">审核状态，必须为*或 0，*表示所有数据，0 表示未审
       核数据</param>
    private static List<HospitalReport> consumeReportByAudit(string
    sbjgbh,DateRangeArg date,char auditState)
    {
        if (!(auditState == '*' || auditState == '0'))
            throw new ArgumentException(" must be * or 0 ","auditState");
        //生成 where 条件
        string predicate = "  ";
        if (auditState == '0')
            predicate = " and (audit no is null) ";
        else if (auditState == '*')
            predicate = "  ";
        //构建 select 语句
        string sql1 = "select sum(a.amt) as amt,sum(a.fee) as fee, "
            +"sum(a.realamt) as realamt,b.yybm from "
            +" (select sum(amt) as amt,sum(fee) as fee ,sum(realamt) as
            realamt,corno,term  "
            +" from bank transaction where "
            +" (time between :date1 and :date2) "+predicate
            +" group by corno,term) a "
            +" inner join institute_dictionary b "
            +" on a.corno=b.corno and a.term=b.term and b.sbjgbh=:sbjgbh  "
            +" group by b.yybm ";
        var oracle = MyDbUtility.oracle;
        //创建命令对象，并添加各个参数
        DbCommand command = oracle.GetSqlStringCommand(sql1);
        oracle.AddInParameter(command, ":sbjgbh", DbType.String, sbjgbh);
        oracle.AddInParameter(command, ":date1", DbType.DateTime,
        date.from);
        oracle.AddInParameter(command, ":date2", DbType.DateTime,
        date.to);
        //执行查询，得到数据读取器，读取数据
        IDataReader reader = oracle.ExecuteReader(command); //执行查询命令
        List<HospitalReport> list = new List<HospitalReport>();
        while (reader.Read())
        {
            //生成报表数据
            HospitalReport item = new HospitalReport();
            item.amt = Convert.ToDouble(reader["amt"]);
            item.fee = Convert.ToDouble(reader["fee"]);
```

```
                item.realamt = Convert.ToDouble(reader["realamt"]);
                item.yybm = Convert.ToString(reader["yybm"]);
                item.sbjgbh = sbjgbh;
                string yymc = "没有对应医院";
                if (!(string.IsNullOrEmpty(item.yybm) || string.
                IsNullOrEmpty(item.sbjgbh)))
                {
                    var h = HospitalDB.getByID(item.yybm, item.sbjgbh);
                    if (h != null)
                        yymc = h.Name;
                }
                item.yymc = yymc;
                list.Add(item);
            }
            reader.Close();
            command.Dispose();
            return list;
        }
        //审核结算汇总统计
        public static List<HospitalReport> auditReport(string sbjgbh,
        DateRangeArg date)
        {
        ...                     //参考光盘源代码
        }
}
```

（2）在业务逻辑层项目 AccountCheckBLL 中添加一个 ReportBLL 类，代码如下：

```
public static class ReportBLL
{
    /// <summary>
    /// 根据社保机构编号、日期范围和报表种类生成报表数据
    /// </summary>
    public static List<HospitalReport> getReport(string sbjgbh,
    DateRangeArg date,ReportType type)
    {
        switch (type)
        {
            case ReportType.ConsumeReport:
                return ReportDB.consumeReport(sbjgbh, date);
            case ReportType.UnauditReport:
                return ReportDB.unauditReport(sbjgbh, date);
            case ReportType.AuditReport:
                return ReportDB.auditReport(sbjgbh, date);
            default:
                throw new ApplicationException("不能识别的报表类型");
        }
    }
}
//报表种类枚举类型
public enum ReportType
{
    ConsumeReport,                                  //消费报表
    UnauditReport,                                  //未审核报表
    AuditReport                                     //已审核报表
}
```

（3）在表现层项目 AccountCheckWeb 中添加一个页面 CompensationStatePage.aspx，页面外观如图 13.30 所示，页面代码如下：

```
<asp:Content  ID="Content2"  ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">
  <h3>定点机构结算情况统计</h3>
   社保机构:                                              <!--社保机构列表-->
   <asp:DropDownList ID="hospitalList"  runat="server">
   </asp:DropDownList>
   日期: <asp:TextBox ID="date1" runat="server"></asp:TextBox>
   至<asp:TextBox  ID="date2" runat="server"></asp:TextBox>
   <asp:RadioButton ID="auditRadio" runat="server" GroupName="group1"
   Text="已结算" Checked="true"/>
   <asp:RadioButton ID="unauditRadio" runat="server" GroupName="group1"
   Text="未结算" />  
   <asp:Button ID="OK" runat="server" Text="统计" OnClientClick="return
   checkInput();" onclick="OK_Click" /> <br />
   <div>
   <asp:GridView ID="grid1" runat="server" CssClass="gridview"
   AutoGenerateColumns="false">
   <Columns>
   <asp:BoundField DataField="yybm" HeaderText="医院编码"
   ItemStyle-Width="100" />
   <asp:BoundField DataField="yymc" HeaderText="医院名称"
   ItemStyle-Width="200"/>
   <asp:BoundField DataField="amt" HeaderText="消费金额"
   DataFormatString="{0:C}" ItemStyle-Width="100"/>
   <asp:BoundField DataField="fee" HeaderText="手续费"
   DataFormatString="{0:C}" ItemStyle-Width="100"/>
   <asp:BoundField DataField="realamt" HeaderText="结算金额"
   DataFormatString="{0:C}" ItemStyle-Width="100"/>
   </Columns>
      <EmptyDataTemplate>         <!--没有数据时显示的提示信息-->
         <span style="color:Red; font-weight:bold;">没有符合条件的数据
         </span>
      </EmptyDataTemplate>
   </asp:GridView>
</div>
</asp:Content>
```

（4）在 CompensationStatePage.aspx 页面“统计”按钮的 Click 事件中，根据用户输入的条件查询数据，并显示在 GridView 中。

```
protected void OK_Click(object sender, EventArgs e)
{
    List<HospitalReport> list=null;
    //得到所查询的日期范围
    DateRangeArg date=DateRangeArg.between2Date(date1.Text, date2.Text);
    string sbjg = hospitalList.SelectedValue;      //社保机构编号
    //根据单选按钮，进行不同查询
    if (auditRadio.Checked)
        list = ReportBLL.getReport(sbjg,date , ReportType.AuditReport);
    else
        list = ReportBLL.getReport(sbjg, date, ReportType.UnauditReport);
    //将查询结果绑定到 GridView 控件
    grid1.DataSource = list;
    grid1.DataBind();
}
```

13.11　小　　结

本章介绍了作者为 A 市人力资源和社会保障局开发的一个软件项目：社保卡结算系统。本项目使用 Oracle 作为后台数据库，数据关系较为复杂，应用程序功能主要包括银行数据导入、费用审核、资金结算、统计报表等功能。

第 14 章　新农合管理系统

新型农村合作医疗（简称新农合）作为造福广大农民的一项重大举措，受到国家和各级政府的高度重视。卫生部要求各地建立起与新农合制度发展相适应、与建设中的国家卫生信息系统相衔接、较为完备和高效的全国新农合信息系统。在各级新农合管理部门、经办机构、定点医疗机构及其他相关部门间建立计算机网络联接，实现网上在线审核结算、实时监控和信息汇总，实现新农合业务管理的数字化、信息化、科学化，提高新农合工作效率和服务水平。

在这种形势下，各级新型农村合作医疗管理部门都开始了信息化建设工作。各软件公司也抓住这一机遇，开发了各种新型农村合作医疗管理信息系统，简称新农合系统。本章将讲解作者所开发的一套新农合系统。

14.1　整体设计思路

新农合系统用户数量多，地域覆盖范围广，功能模块复杂。在开发此系统以前，需要详细了解新农合系统工作流程和用户需求。本节将介绍新农合系统要实现的主要功能及系统的整体结构。

14.1.1　新农合业务流程

新型农村合作医疗，简单来说就是给农村实行的医疗保险。保险资金的筹集包括 3 部分：省财政拨款、县财政拨款和农民个人缴费。例如，每个农民参加新农合，每年需要缴纳 20 元，县级财政拨款 50 元，省级财政拨款 50 元，投保资金一共是 120 元。所有农民的投保资金汇总到一起，就形成了新农合基金。当参合农民生病住院时，就可以按照相关政策报销一定比例的费用，而这些报销费用从新农合基金中支取。

在新农合刚开始实施阶段，还没有实现信息化，农民住院后，需要拿着相关证明（如住院费用发票、病历复印件等）到合作医疗管理办公室进行审核和报销。这种报销方式周期长，成本高，效率低。为了使新农合政策的实施更加顺畅，各地都先后对新农合业务实行了信息化管理。

实行信息化管理以后，新农合系统实现全市甚至全省联网，各种数据（如病人信息、药品信息、医生信息等）都可以实现共享。农民出院结算时，根据身份证号，即可得到农民的参合信息（如是否参合、本年度可报销金额等），新农合系统会自动计算报销金额，并从住院费用中扣除报销金额，农民只需支付其余的金额给医院。例如，某参合农民住院共

花费 3000 元，可报销 1000 元，则此病人只需交给医院 2000 元，其余 1000 元从新农合基金中支付。

参合农民出院时，实际缴纳的出院费用低于农民的真实费用，其差额由新农合基金支付。这些资金需要医院到新农合办公室领取。每隔一段时间，医院需要到新农合办公室进行结算。此时新农合系统会计算该时间段内共有多少参合农民在此医院住院，花费了多少钱，应该由新农合报销多少钱，然后将报销金额支付给医院。至此，医院收到了病人的全部住院费用。根据以上分析，可以得到新农合系统业务流程图如图 14.1 所示。

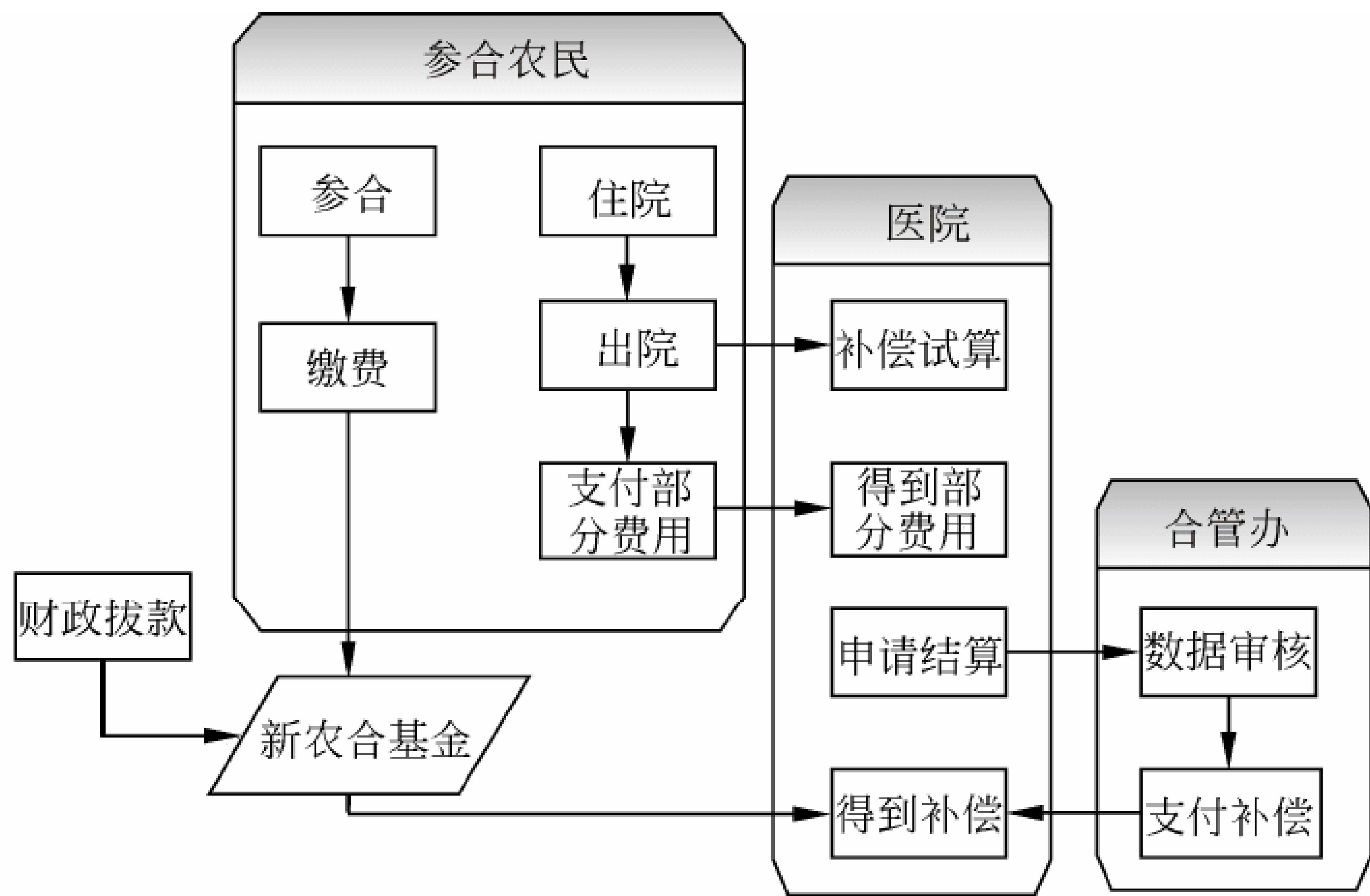


图 14.1　新农合业务流程图

14.1.2　系统功能模块

新农合系统的用户主要包括两类：一类是各级新农合管理部门，如市、县（区）卫生局，乡（镇）合作医疗管理办公室，另一类是定点医疗机构，即新农合定点医院。新农合管理部门使用新农合系统进行基础数据管理、费用审核等操作，定点医疗机构使用新农合系统进行住院费用结算、药品字典管理等操作。整个系统功能如图 14.2 所示。

1．数据字典管理

此模块用于维护系统中用到的基础数据，如职业、民族、行政区划等。

2．参合档案管理

此模块用于维护参合农民和家庭信息。

3．参合缴费

此模块实现参合农民缴费和单据打印功能。

4．药品目录管理

此模块用于管理基本用药目录，只有在目录范围之内的药品才能报销，而且不同药品

有不同的报销比例。

5．报销政策管理

此模块用于设置报销政策，如起报线、分段报销比例、单病种报销金额等。

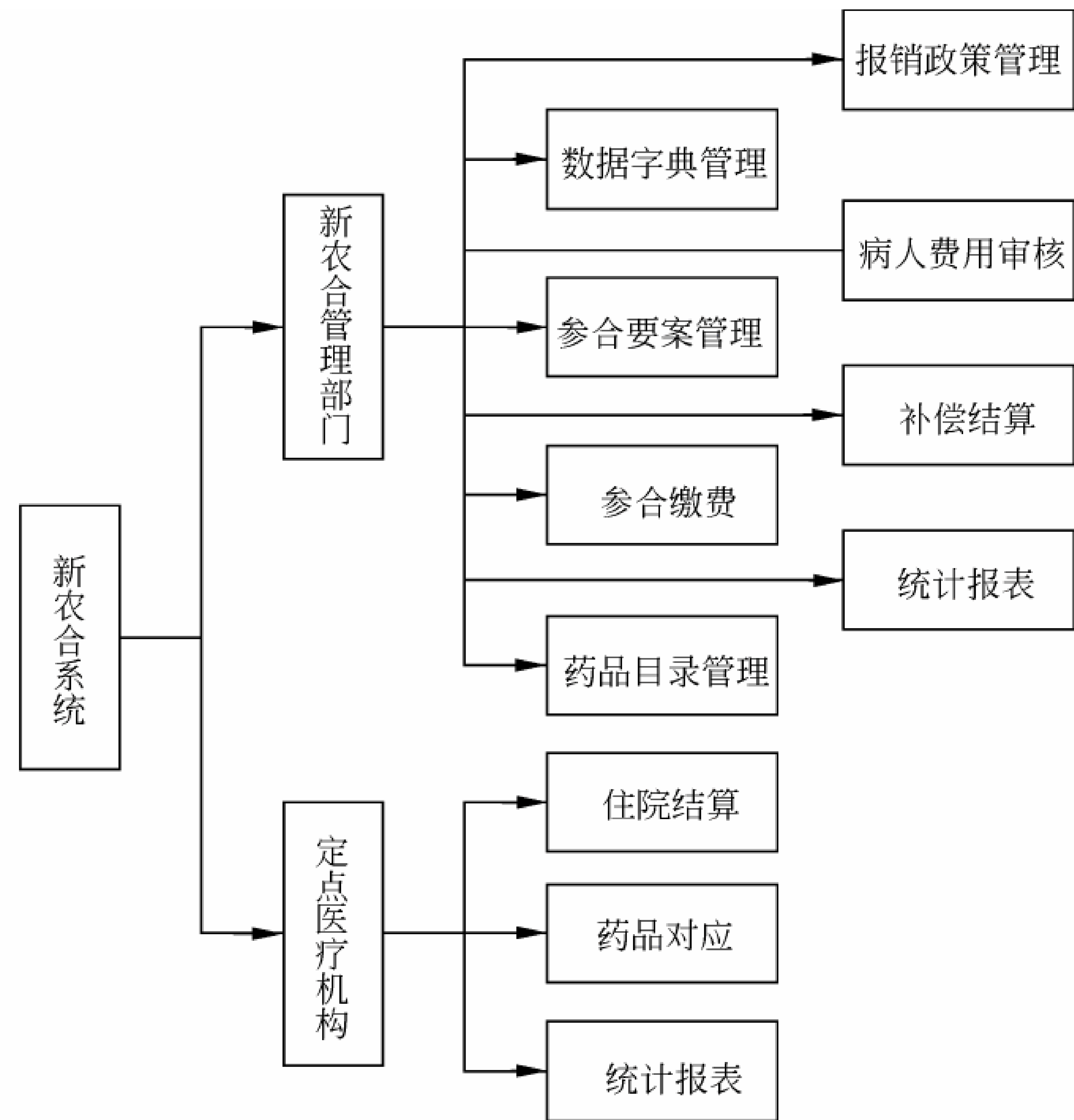


图 14.2　新农合系统功能模块图

6．病人费用审核

此模块用于审核病人住院费用，检查是否存在诸如不合理使用昂贵药品、虚假住院费用等情况。

7．补偿结算

根据病人在医院的住院费用与医院进行补偿结算。

8．统计报表

此模块用于生成各种统计报表。

9．住院结算

参合病人出院时，医疗机构需要将数据上传到新农合数据库中，并计算病人需要实际支付的住院费用金额和新农合报销金额。

10．药品对应

由于各个医院药品编码不一致，各医院需要将各自的药品对应到新农合系统中的相应

药品，以实现数据传输和共享。

14.1.3　数据库结构

新农合系统数据结构较为复杂，数据库中共有大约 50 个表，如图 14.3 所示。在了解新农合系统的细节功能以前，不容易理解数据库的结构，此处不详细介绍数据库结构，等到实现某个功能时再具体介绍相关表结构。

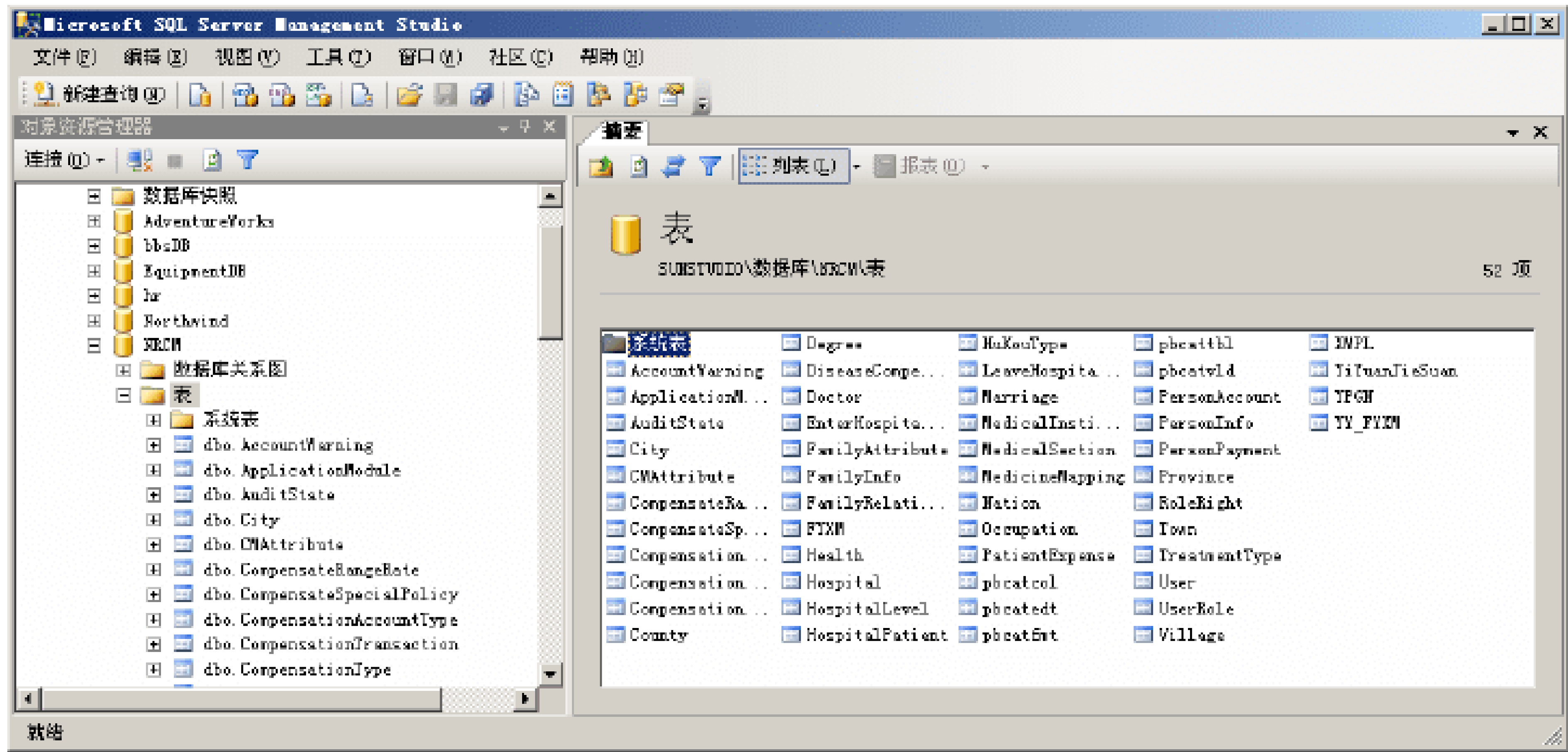


图 14.3　新农合数据库中的表

14.1.4　搭建项目框架

新农合系统采用多层结构设计，使用实体框架实现数据访问功能。整个新农合项目解决方案中包含以下几个项目：实体框架层、数据访问层、业务逻辑层、Web 表现层、公共类库、单元测试项目、短信通知。搭建整个解决方案框架的步骤如下。

（1）创建一个空白解决方案 NRCM，由于解决方案中项目较多，为了使项目结构更加清晰，在解决方案中添加两个解决方案文件夹：DLL 和 Test，分别用于包含类库项目和测试项目。

（2）在解决方案 DLL 文件夹中，添加一个类库项目 Nrcm.Entity 作为实体框架层。

（3）在解决方案 DLL 文件夹中，添加一个类库项目 Nrcm.Dal 作为数据访问层。

（4）在解决方案 DLL 文件夹中，添加一个类库项目 Nrcm.Bll 作为业务逻辑层。

（5）在解决方案 DLL 文件夹中，添加一个类库项目 Nrcm.Common，此项目中包含其他各个项目都会用到的公共类。

（6）在解决方案中添加一个 ASP.NET Web 应用程序项目 Nrcm.Web，作为表现层。

（7）在解决方案 Test 文件夹中添加一个类库项目 DalTest，作为数据访问层的单元测试项目。

（8）在解决方案 Test 文件夹中添加一个类库项目 BllTest，作为业务逻辑层的单元测试项目。

（9）在各个项目之间添加引用关系，表现层 Nrcm.Web、业务逻辑层 Nrcm.Bll、数据访问层 Nrcm.Dal 都引用实体框架层 Nrcm.Entity 和公共类库 Nrcm.Common，业务逻辑层

Nrcm.Bll 引用数据访问层 Nrcm.Dal，表现层 Nrcm.Web 引用业务逻辑层 Nrcm.Bll，测试项目引用被测试项目。整个解决方案结构及各个项目之间的关系如图 14.4 所示。

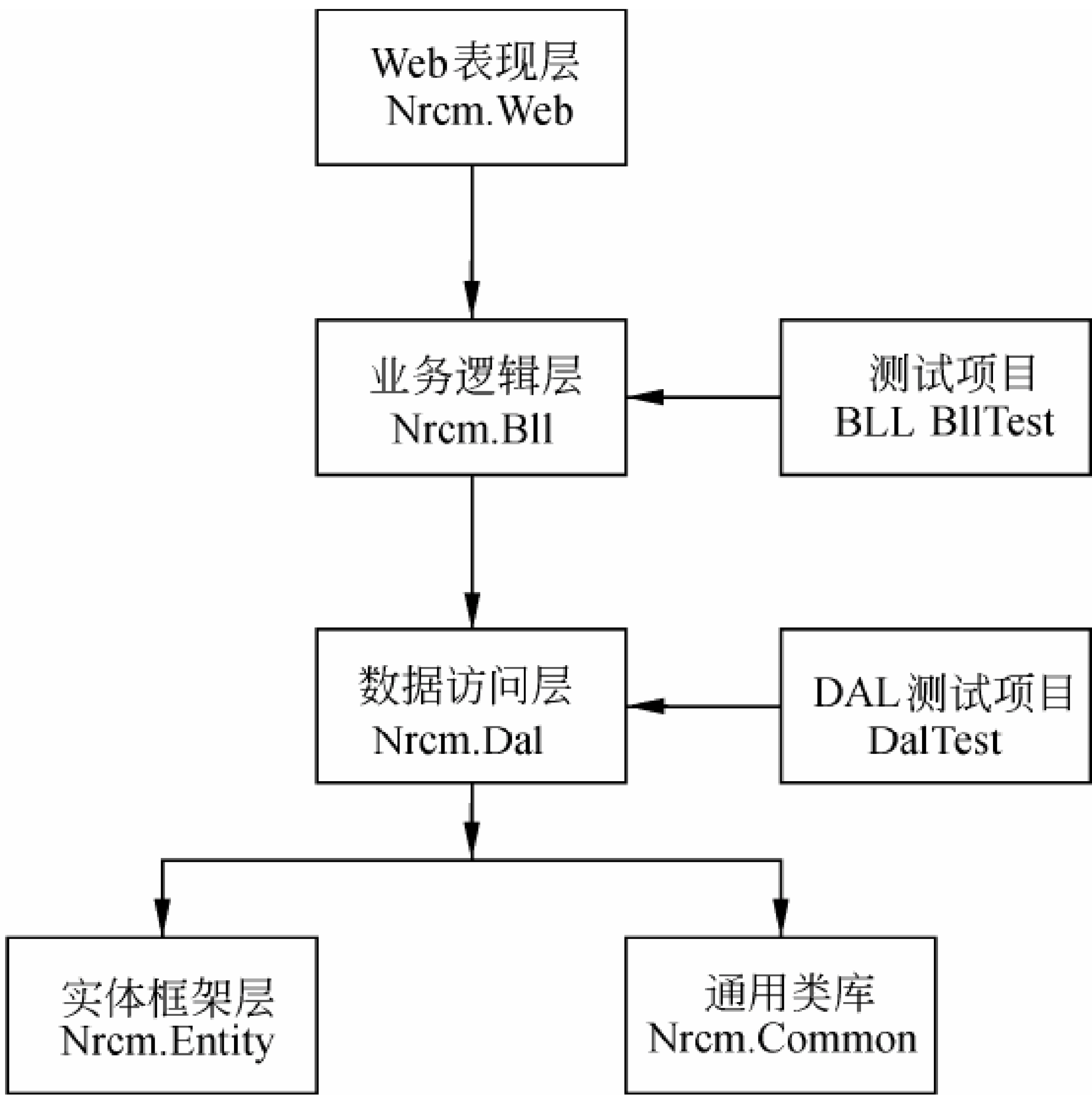


图 14.4　新农合系统项目结构

14.2　母版页设计

新农合系统的页面基本由 4 部分组成：页面顶部的 LOGO 和工具栏，页面左侧的导航栏，页面右侧的页面正文，页面底部的页脚。由于新农合系统功能模块较多，为了条理清楚地组织所有模块，系统使用页面顶部的工具栏和页面左侧的导航栏进行功能导航。页面顶部的工具栏作为第一级菜单，页面左侧导航栏作为第二级菜单。一个典型的新农合系统的页面结构如图 14.5 所示。



图 14.5　新农合系统典型页面结构

在图 14.5 所示的新农合页面中，顶部的页头和底部的页脚对于所有页面都是相同的，因此可以将其放于母版页中。虽然大多数页面左侧都有导航栏，但是也存在个别没有左侧导航栏的页面，而且对于不同的一级菜单（工具栏上的按钮），左侧导航栏的内容也不相同。为了同时兼顾通用和定制，本系统采用两级母版页，即主母版页中仅包含页头页脚内容，对于不同的功能模块（如报销政策管理模块），再基于主母版页创建一个二级母版页，作为该模块中所有页面的母版页。

14.2.1 天气预报用户控件

在新农合系统母版页的顶部显示了今天的天气情况，该数据来源于互联网上发布的天气预报 Web 服务。本系统使用一个用户控件 Weather.ascx 封装了天气预报功能。创建 Weather.ascx 用户控件的步骤如下。

（1）在表现层项目 Nrcm.Web 中添加一个 Web 服务引用，地址为 http://www.ayandy.com/Service.asmx，此 Web 服务能够免费提供全国各地的天气预报数据。

（2）在表现层项目 Nrcm.Web 中添加一个类 WeatherServcie，封装对天气预报 Web 服务的调用。由于天气预报是一种相对稳定的数据，没有必要频繁调用 Web Service 浪费性能，本系统每隔 10 分钟刷新一次天气预报数据。在代码中判断当前时间距上次调用 Web Service 刷新天气信息是否已达到或超过 10 分钟，如果未到 10 分钟则不调用 Web Service，而是使用上次得到的天气数据。

```
internal class WeatherService
{
    internal const string city = "北京";
    internal static DateTime lastGetTime;        //上次获得天气信息的时间
    internal static bool success = false;        //是否成功获得天气信息
    private static WeatherInfo weatherResult=new WeatherInfo();
                                                 //天气信息
    /* 根据城市名称获得天气情况。
     * 调用方法如下：输入参数 theCityName 城市中文名称，如深圳，北京；
     * theDayFlag 指定是当天(1),明天(2)或后天(3)，可查询未来三天的天气情况；
     * 返回数据： String[7]一个一维数值，共有 7 个元素，
     *           从[1]到[6]分别表示城市，天气，温度，风向，日期，天气图标地址。
     */
    internal static void getWeather()
    {
        //每隔 10 分钟刷新一次天气信息，如果当前时间距上次刷新不到 10 分钟，则直接返回
        if (success && lastGetTime.AddMinutes(10) < DateTime.Now)
            return ;
        Console.WriteLine("get weather ...");
        lastGetTime = DateTime.Now;
        try
        {
            var service = new weather.Service();
                                           //创建 Web Service 客户端代理
            string [] result=service.getWeatherbyCityName
                (city, weather.theDayFlagEnum.Today);
                                           //调用 Web Service 得到天气信息
            success = true;
            buildResult(result);
        }
```

```
        catch
        {
            weatherResult.clear();
            weatherResult.weather = "未能获取天气信息";
            success = false;
        }
    }
    //将 Web Service 返回的 string 数组绑定到 WeatherInfo 对象
    private static void buildResult(string[] result)
    {
        weatherResult.weather = result[2];
        weatherResult.temprature = result[3];
        weatherResult.wind = result[4];
        weatherResult.date = result[5];
        weatherResult.imgUrl = result[6];
    }
    public static WeatherInfo Weather
    {
        get { return weatherResult; }
    }
}
/// <summary>
/// 此类用于描述天气信息
/// </summary>
internal class WeatherInfo
{
    public string weather = "";                     //天气（晴、阴、多云等）
    public string temprature = "";                  //温度
    public string wind = "";                        //风向
    public string date = "";                        //日期
    public string imgUrl = "";                      //天气图片 url
    internal void clear()                           //清除天气信息
    {
        weather = "";
        temprature = "";
        wind = "";
        date = "";
        imgUrl = "";
    }
}
```

（3）在表现层项目 Nrcm.Web 中新建一个 UserControls 文件夹，在此文件夹中添加一个新的用户控件 Weather.ascx。

（4）在 Weather.ascx 用户控件中用一个 table 控件，显示城市名称和天气信息（文字描述和图片）。代码如下：

```
<table>
<tr>
<td><asp:Label ID="cityName" runat="server" Text="城市名称"></asp:Label><br
/>天气</td>
<td><asp:Image    ID="weatherImage"    runat="server"    Height="32px"
AlternateText="天气" /></td>
<td>
<asp:Label ID="weatherDesc" runat="server" Text="天气情况"></asp:
Label></td>
</tr>
</table>
```

（5）在 Weather.ascx 用户控件的 Page_Load 事件中调用 WeatherService 类，得到天气信息并显示于页面上。

```
protected void Page Load(object sender, EventArgs e)
{
    if(!IsPostBack)
        cityName.Text = WeatherService.city;
    showWeather();
}
private void showWeather()
{
    WeatherService.getWeather();
    weatherDesc.Text = WeatherService.Weather.weather + "<br/>"
        + WeatherService.Weather.temprature;        //显示天气文字描述
    weatherImage.ImageUrl = WeatherService.Weather.imgUrl; //显示天气图片
}
```

14.2.2　页头用户控件

新农合系统的页面头部用于显示系统名称、当前时间、当前用户、天气信息、工具栏，其中天气信息使用 14.2.1 节中所创建的 Weather.ascx 显示。新农合页面头部也被封装为一个用户控件 Header.ascx，控件外观如图 14.6 所示。



图 14.6　Header.ascx 用户控件外观

（1）在表现层项目 Nrcm.Web 中添加一个用户控件 Header.ascx，页面代码如下：

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="Header.ascx.
cs" Inherits="Nrcm.Web.UserControls.Header" %>
<%--注册天气预报用户控件--%>
<%@ Register src="Weather.ascx" tagname="Weather" tagprefix="uc1" %>
<div style="background:#f2f2fe; width:1000px; text-align:left;">
<%--工具栏上面的 div--%>
<div style="float:left">
<img src="../images/title.png" alt="" />
<span>
当前日期：<%=DateTime.Now.ToLongDateString()%>
当前用户：
 <%= Nrcm.Web.HttpCode.WebUtility.currentUser == null ?
    "未知用户":Nrcm.Web.HttpCode.WebUtility.currentUser.UserName.Trim()%>
【<a href="../LogoutPage.aspx">退出登录</a>】
</span>
</div>
   <uc1:Weather ID="Weather1" runat="server" /> <%--天气预报--%>
</div>
<div id="NavMenu">  <%--工具栏--%>
<div class="ImageMenu"><img src="../images/user.png" alt="农民档案"/><br/>
<a href="../People/FamilyPage.aspx">农民参合档案</a></div>
<div class="ImageMenu"><img src="../images/medicine.png" alt="药品目录"
/><br/>
<a href="../Medicine/MedicinePage.aspx">药品检查目录</a></div>
```

```
<div class="ImageMenu"><img src="../images/hospital.png" alt="医疗机构"
/><br/>
<a href="../Hospital/HospitalPage.aspx">定点医疗机构</a></div>
<div class="ImageMenu"><img src="../images/policy.png" alt="报销政策"
/><br/>
<a href="../Policy/CopensateRatePage.aspx">报销政策管理</a></div>
<div class="ImageMenu"><img src="../images/dictionary.png" alt="字典维护"
/><br/>
<a href="../Dictionary/DictionaryPage.aspx">数据字典维护</a></div>
<div class="ImageMenu"><img src="../images/lock.png" alt="用户权限" /><br/>
<a href="../UserRight/ChangePassPage.aspx">用户权限管理</a></div>
</div>
```

（2）Header 用户控件工具栏中“退出登录”按钮链接到 LogoutPage.aspx 页面，此页面将清空 Session 中保存的登录数据。页面代码如下：

```
<form id="form1" runat="server">
    <div id="main">
    <h3>你已经退出新农合管理系统，再见。</h3>
     点击<a href="Login.aspx">这里</a>重新登录<br />
    </div>
</form>
protected void Page_Load(object sender, EventArgs e)
{
    Nrcm.Web.HttpCode.WebUtility.currentUser = null;
}
```

14.2.3　母版页

正如本节开始部分所述，新农合系统采用两级母版页，其中主母版页内容较为简单，仅包含页头控件、页脚控件和页面中间的一个内容占位符（ContenPlaceHolder）控件。主母版页 Nrcm.Master 页面代码如下：

```
<div id="main">
   <form id="form1" runat="server">
   <div>
      <uc1:Header ID="Header1" runat="server" /> <%--页头控件--%>
   </div>
   <div>
      <asp:ContentPlaceHolder ID="content" runat="server">
      <%--内容占位符--%>
      </asp:ContentPlaceHolder>
    </div>
    <div>
      <uc2:Footer ID="Footer1" runat="server" />  <%--页脚控件--%>
   </div>
   </form>
</div>
```

14.3　基础数据管理

新农合系统用到一些基础数据，包括在很多管理信息系统中都会用到的数据字典如职

业编码、民族代码、行政区划等，还包括新农合系统所特有的一些数据，如分段报销比例、报销参数等。新农合管理系统需要设计一些页面以允许用户修改这些数据。

14.3.1　数据字典管理

管理信息系统的数据库中一些属性的描述通常使用编码而不是文字，例如，职业、民族、健康状况、学历等，新农合系统也是如此。这些数据字典通常允许用户添加、删除和修改。此处以职业为例说明如何设计实现简单数据管理页面。职业数据保存在数据库的 Occupation 表中，有两个字段职业编号 ID 和职业名称 Name，两字段都是字符类型。

职业管理页面 JobPage.aspx 布局分为上下两部分，上面一部分用一个 GridView 显示目前已经存在的职业列表，下面用一个 DetailsView 添加新职业，DetailsView 的默认显示模式为插入模式，页面上使用 SqlDataSource 作为数据源。JobPage.aspx 页面设计视图如图 14.7 所示。

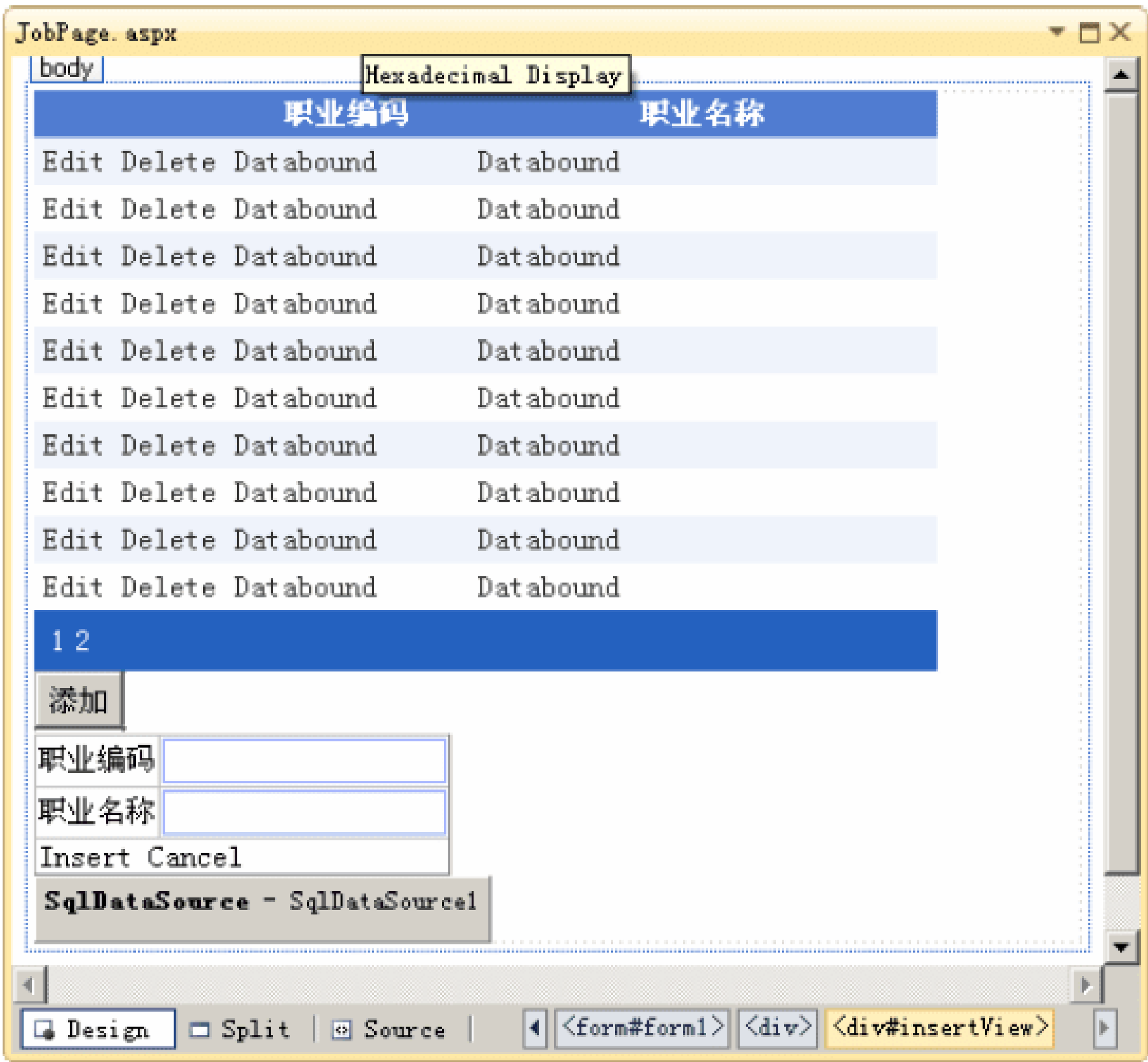


图 14.7　职业管理页面设计视图

JobPage.aspx 页面代码如下：

```
<form id="form1" runat="server">
  <div>
  <%--此 GridView 用于显示职业列表--%>
     <asp:GridView ID="GridView1" runat="server" AllowPaging="True"
         AutoGenerateColumns="False" DataKeyNames="ID" DataSourceID=
         "SqlDataSource1">
         <Columns>
             <%--GridView 的命令列，显示编辑和删除命令--%>
             <asp:CommandField ShowDeleteButton="True" ShowEditButton=
             "True" />
             <asp:BoundField DataField="ID" HeaderText="职业编码" ReadOnly=
             "True"
                 SortExpression="ID">
             <HeaderStyle Width="100px" />
             </asp:BoundField>
             <asp:BoundField DataField="Name" HeaderText="职业名称"
```

```
            SortExpression="Name">
            <HeaderStyle Width="200px" />
            </asp:BoundField>
        </Columns>
    </asp:GridView>
    <%--添加按钮为客户端按钮，避免产生不必要的页面回发--%>
   <input type="button" value="添加" id="addButton"/>
   <div id="insertView">
   <%--DetailsView 用于添加新的职业，默认模式为插入模式--%>
    <asp:DetailsView ID="DetailsView1" runat="server"
        AutoGenerateRows="False" DataKeyNames="ID" DataSourceID=
        "SqlDataSource1"
        DefaultMode="Insert">
        <Fields>
            <asp:BoundField DataField="ID" HeaderText="职业编码" ReadOnly=
            "True"
                SortExpression="ID" />
            <asp:BoundField DataField="Name" HeaderText="职业名称"
            SortExpression="Name" />
            <asp:CommandField ShowInsertButton="True" />
        </Fields>
    </asp:DetailsView>
    </div>
    <%--鉴于职业数据非常简单，没有必要三层结构，直接使用 SqlDataSource 实现数据访问
    --%>
    <asp:SqlDataSource ID="SqlDataSource1" runat="server"
        ConnectionString="<%$ ConnectionStrings:NRCM %>"
        DeleteCommand="DELETE FROM [Occupation] WHERE [ID] = @ID"
        InsertCommand="INSERT INTO [Occupation] ([ID], [Name]) VALUES (@ID,
        @Name)"
        SelectCommand="SELECT [ID], [Name] FROM [Occupation]"
        UpdateCommand="UPDATE [Occupation] SET [Name] = @Name WHERE [ID] =
        @ID">
        <DeleteParameters>
            <asp:Parameter Name="ID" Type="String" />
        </DeleteParameters>
        <UpdateParameters>
            <asp:Parameter Name="Name" Type="String" />
            <asp:Parameter Name="ID" Type="String" />
        </UpdateParameters>
        <InsertParameters>
            <asp:Parameter Name="ID" Type="String" />
            <asp:Parameter Name="Name" Type="String" />
        </InsertParameters>
    </asp:SqlDataSource>
</div>
</form>
```

默认情况下页面底部的添加区域不可见，只有当用户单击“添加”按钮时才会显示。“添加”按钮是一个 html 浏览器端按钮，不会引起服务器端回发，在浏览器端通过 JavaScript 脚本显示添加职业的区域。

```
<script type="text/javascript" >
    $(function () {
        $('#insertView').hide();                  //隐藏添加区域
        $('#addButton').click(function () {      //单击 add 按钮时显示添加区域
            $('#insertView').show();
        });
</script>
```

14.3.2　行政区划管理

新农合管理系统需要对行政区划（主要是乡镇和村）数据进行管理。各级行政区划的维护页面大致相同，此处以维护乡镇数据为例说明页面如何实现。数据库中乡镇表 Town 包含以下 3 个字段。

- ID：nvarchar(10)类型，乡镇编号，主键。
- Name：nvarchar(20)类型，乡镇名称。
- ParentID：上级（即所属区县）ID，外键关联到区县表 County。

（1）在实体框架项目 Nrcm.Entity 中添加行政区划数据的实体模型 DictionaryContext，数据模型内容如图 14.8 所示。

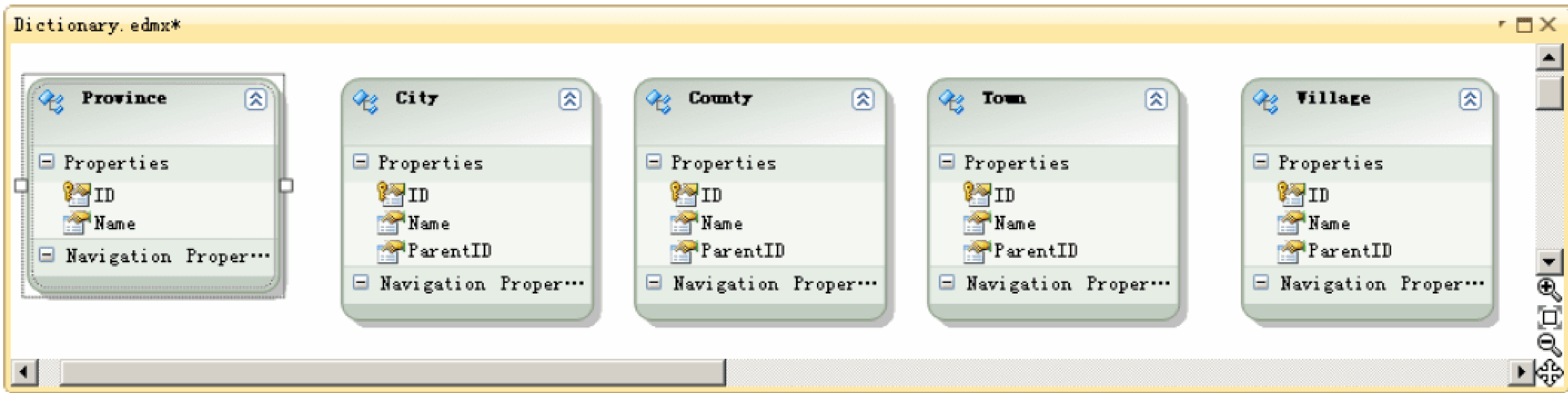


图 14.8　行政区划数据模型

（2）在处理行政区划数据时，经常需要得到一个行政区划的整个层次结构，例如，处理村级数据时，需要知道村所属的省、市、县、乡。为了描述这个层次结构，在业务逻辑层 Nrcm.Bll 项目中添加一个 DistrictHierarchy 类。

```
//行政区划层次数据
public class DistrictHierarchy
{
    public string id=null;                        //当前行政区划 ID
    public string name=null;                      //当前行政区划名称
    public DistrictLevel level;                   //当前行政区划级别
    public DistrictHierarchy parent=null;         //上级行政区划层次
}
//行政区域级别枚举类型
public enum DistrictLevel
{
    Province, City, County, Town, Village
}
```

（3）在业务逻辑层项目 Nrcm.Bll 中添加一个 DistrictUtility 类，此类主要功能为获取各级行政划的层次结构。

```
public class DistrictUtility
{
    //初始化 5 个 BLL 对象
    CityBLL citybll = new CityBLL();
    ProvinceBLL provincebll = new ProvinceBLL();
    CountyBLL countybll = new CountyBLL();
    TownBLL townbll = new TownBLL();
```

```
    VillageBLL villagebll = new VillageBLL();
    //以下为函数字典，通过行政级别得到一个获得该行政级别层次结构的函数
    Dictionary<DistrictLevel, Func<string, DistrictHierarchy>>
    functionTable;
    public DistrictUtility()
    {
        functionTable = new Dictionary<DistrictLevel, Func<string,
        DistrictHierarchy>>();
        functionTable.Add(DistrictLevel.Province, getProvinceHierarchy);
        functionTable.Add(DistrictLevel.City, getCityHierarchy);
        functionTable.Add(DistrictLevel.County, getCountyHierarchy);
        functionTable.Add(DistrictLevel.Town, getTownHierarchy);
        functionTable.Add(DistrictLevel.Village, getVillageHierarchy);
    }
    //根据行政区域编号和级别获得其层次结构
    public DistrictHierarchy getHierarchy(string districtID, DistrictLevel
    level)
    {
        return functionTable[level](districtID); //调用函数字典中的相应函数
    }
    //得到省级行政层次结构
    private DistrictHierarchy getProvinceHierarchy(string id)
    {
        var p = provincebll.getProvinceByID(id);
        if (p == null)
            throw new ApplicationException("没有找到与此对应的行政区域。");
        return new DistrictHierarchy()
        { id = p.ID, name = p.Name, parent = null, level=DistrictLevel.
        Province };
    }
    //得到市级行政层次结构
    private DistrictHierarchy getCityHierarchy(string id)
    {
        City current = citybll.getCityById(id);
        if (current == null)
            throw new ApplicationException("没有找到与此对应的行政区域。");
        DistrictHierarchy d = new DistrictHierarchy()
        { id = current.ID, name = current.Name, level=DistrictLevel.City };
        d.parent = getProvinceHierarchy(current.ParentID);
                                            //得到其上一级层次结构
        return d;
    }
    //得到县级行政层次结构
    private DistrictHierarchy getCountyHierarchy(string id)
    {
        var current = countybll.getCountyById(id);
        if (current == null)
            throw new ApplicationException("没有找到与此对应的行政区域。");
        DistrictHierarchy d = new DistrictHierarchy()
        { id = current.ID, name = current.Name, level = DistrictLevel.
        County };
        d.parent = getCityHierarchy(current.ParentID);
        return d;
    }
    //得到乡镇级行政层次结构
    private DistrictHierarchy getTownHierarchy(string id)
    {
        var current = townbll.getTownById(id);
        if (current == null)
```

```
            throw new ApplicationException("没有找到与此对应的行政区域。");
        DistrictHierarchy d = new DistrictHierarchy()
        { id = current.ID, name = current.Name, level = DistrictLevel.Town };
        d.parent = getCountyHierarchy(current.ParentID);
        return d;
    }
    //得到村级行政层次结构
    private DistrictHierarchy getVillageHierarchy(string id)
    {
        var current = villagebll.getVillageById(id);
        if (current == null)
            throw new ApplicationException("没有找到与此对应的行政区域。");
        DistrictHierarchy d = new DistrictHierarchy()
        { id = current.ID, name = current.Name, level = DistrictLevel.
        Village };
        d.parent = getTownHierarchy(current.ParentID);
        return d;
    }
}
```

（4）在数据访问层项目 Nrcm.Dal 中添加一个类 TownDal，实现乡镇数据的增删改查功能。

```
//乡镇数据访问类
public class TownDAL:ITownDAL
{
    /// <summary>
    /// 得到指定区县中的乡镇列表
    /// </summary>
    /// <param name="countyId">区县 ID</param>
    /// <param name="arg">分页参数</param>
    /// <returns>得到的乡镇列表</returns>
    public List<Town> getTownsInCounty(string countyId, PageDataArgument
arg)
    {
        using (DictionaryContext context = new DictionaryContext())
        {
            //编写 LINQ 查询代码
            var query = (from c in context.Town
                         where c.ParentID.Trim() == countyId
                         orderby c.ID select c);
            if (arg.refreshCount)                     //是否刷新总数
                arg.count = query.Count();
            //根据分页参数得到指定页的数据
            var list = query.Skip(arg.pageIndex * arg.pageSize).Take(arg.
            pageSize).ToList();
            EntityUtility.detachEntity(context, list);
            return list;
        }
    }
    //根据编号得到乡镇数据
    public Town getTownById(string id)
    {
        using (DictionaryContext context = new DictionaryContext())
        {
            return EntityUtility.selectOne<Town>(context.Town, t => t.ID ==
            id);
        }
    }
```

```
    //根据乡镇名称得到某区县中的乡镇信息（一个区县中不会存在同名乡镇）
    public Town getTownByNameInCounty(string name,string countyId)
    {
        using (DictionaryContext context = new DictionaryContext())
        {
            return EntityUtility.selectOne<Town>(context.Town, t => t.Name
            == name && t.ParentID == countyId);
        }
    }
    //添加乡镇
    public int addTown(Town town)
    {
        using (DictionaryContext context = new DictionaryContext())
        {
            context.AddToTown(town);
            int n = context.SaveChanges();
            EntityUtility.detachEntity(context, town);
            return n;
        }
    }
    //根据乡镇编号删除乡镇
    public int deleteTown(string id)
    {
        using (DictionaryContext context = new DictionaryContext())
        {
            var town = (from t in context.Town
                        where t.ID == id select t).FirstOrDefault();
            if (town == null) return 0;
            context.DeleteObject(town);
            return context.SaveChanges();
        }
    }
    //修改乡镇数据
    public int updateTown(Town town)
    {
        using (DictionaryContext context = new DictionaryContext())
        {
            return EntityUtility.update(context, town);
        }
    }
}
```

（5）在业务逻辑层 Nrcm.Bll 中添加一个 TownBll 类，由于新农合系统中没有与乡镇相关的业务逻辑，所以 TownBll 类只是调用 TownDal 类的相应方法，此处省略其代码。

（6）在表现层项目 Nrcm.Web 中添加一个用户控件 SelectCounty.ascx，此控件的功能是让用户选择一个区县，这个功能在整个新农合系统中用到多次，包括在添加和修改乡镇时也需要选择所属区县。为了实现复用，将选择区县的功能封装成一个用户控件。此控件中包含 3 个级联下拉列表，分别表示省、市和区县，页面代码如下：

```
<asp:DropDownList ID="province" runat="server" AutoPostBack="true"
   DataValueField="ID" DataTextField="Name"
   onselectedindexchanged="province SelectedIndexChanged">
</asp:DropDownList>
<asp:DropDownList ID="city" runat="server" AutoPostBack="true"
   DataValueField="ID" DataTextField="Name"
   onselectedindexchanged="city SelectedIndexChanged">
</asp:DropDownList>
<asp:DropDownList ID="county" runat="server"  DataValueField="ID"
```

```
DataTextField="Name">
</asp:DropDownList>
```

（7）在用户控件 SelectCounty.ascx 的 Page_Load 事件中，加载所有省份数据。

```
bool provinceLoaded = false;            //省级数据是否已经加载
protected void Page Load(object sender, EventArgs e)
{
    if (!IsPostBack)
        initProvinces();
}
private void initProvinces()
{
    if (provinceLoaded)  return;        //如果数据已经被加载则返回
    provinceLoaded = true;
    //得到所有省份数据并绑定到下拉列表控件
    var list = new ProvinceBLL().getAllProvinces(PageDataArgument.
    allData);
    province.DataSource = list;
    province.DataBind();
    //在下拉列表中添加一个"请选择"项
    province.Items.Insert(0, new ListItem("--请选择--", "-1"));
}
```

（8）在用户控件 SelectCounty.ascx 省级下拉列表的 SelectedIndexChanged 事件中，加载并填充所选择省份的所有市级数据。

```
protected void province SelectedIndexChanged(object sender, EventArgs e)
{
    populateCity();
}
private void populateCity()
{
    string pid=province.SelectedValue;
    DropDownList control = city;
    if (string.IsNullOrEmpty(pid)) return;
    if (pid == "-1") return;
    List<City> list = new CityBLL().getCitiesInProvince(pid,
    PageDataArgument.allData);
    control.DataSource = list;
    control.DataBind();
    ListItem item = new ListItem("--请选择--", "-1");
    control.Items.Insert(0, item);
}
```

（9）在用户控件 SelectCounty.ascx 城市下拉列表的 SelectedIndexChanged 事件中，加载并填充所选择城市的所有区县数据。

```
protected void city SelectedIndexChanged(object sender, EventArgs e)
{
    populateCounty();
}
private void populateCounty()
{
    string pid = city.SelectedValue;
    if (string.IsNullOrEmpty(pid)) return;
    if (pid == "-1") return;
    var control = county;
    List<County> list = new CountyBLL().getCountiesInCity(pid, new
```

```
    PageDataArgument(0, 300, false));
    control.DataSource = list;
    control.DataBind();
    ListItem item = new ListItem("--请选择--", "-1");
    control.Items.Insert(0, item);
}
```

（10）在用户控件 SelectCounty.ascx 中添加一个属性 selectedCounty，用于获取和设置当前选中的区县 ID。当读取该属性时，返回区县下拉列表中所选中的值。当设置该属性时，要设置省、市、县各个下拉列表的当前选中项的值，必要时还需要重新加载数据。

```
//获取或设置当前选中的区县 ID
public string selectedCounty
{
    get
    {
        if (county.SelectedIndex > 0)
            return county.SelectedValue;
        else
            return null;
    }
    set
    {
    //当设置此属性（当前选中的区县 ID）时，需要在省级、市级和县级下拉列表中都显示正确
      数据
        //如果省份未加载则加载，则先加载省份列表
        if (!(provinceLoaded || IsPostBack))
            initProvinces();
        string id = value;
        //得到行政区划层次
        var hierarchy = new DistrictUtility().getHierarchy(id,
        DistrictLevel.County);
        ListItem item=null;
        //判断当前所选中的省份是否正确，否则改变
        if (hierarchy.parent.parent.id != province.SelectedValue)
        {
            item = province.Items.FindByValue(hierarchy.parent.parent.
            id);
            if (item != null) item.Selected = true;
            populateCity();
        }
        //判断当前选中的市级行政区划是否正确，如果不正确则修改
        if (hierarchy.parent.id != city.SelectedValue)
        {
            item = city.Items.FindByValue(hierarchy.parent.id);
            if (item != null) item.Selected = true;
            populateCounty();
        }
        //选中当前县
        item = county.Items.FindByValue(id);
        if (item != null) item.Selected = true;
    }
}
```

（11）在表现层项目 Nrcm.Web 中添加一个页面 TownPage.aspx，用于显示乡镇列表。页面布局如图 14.9 所示。

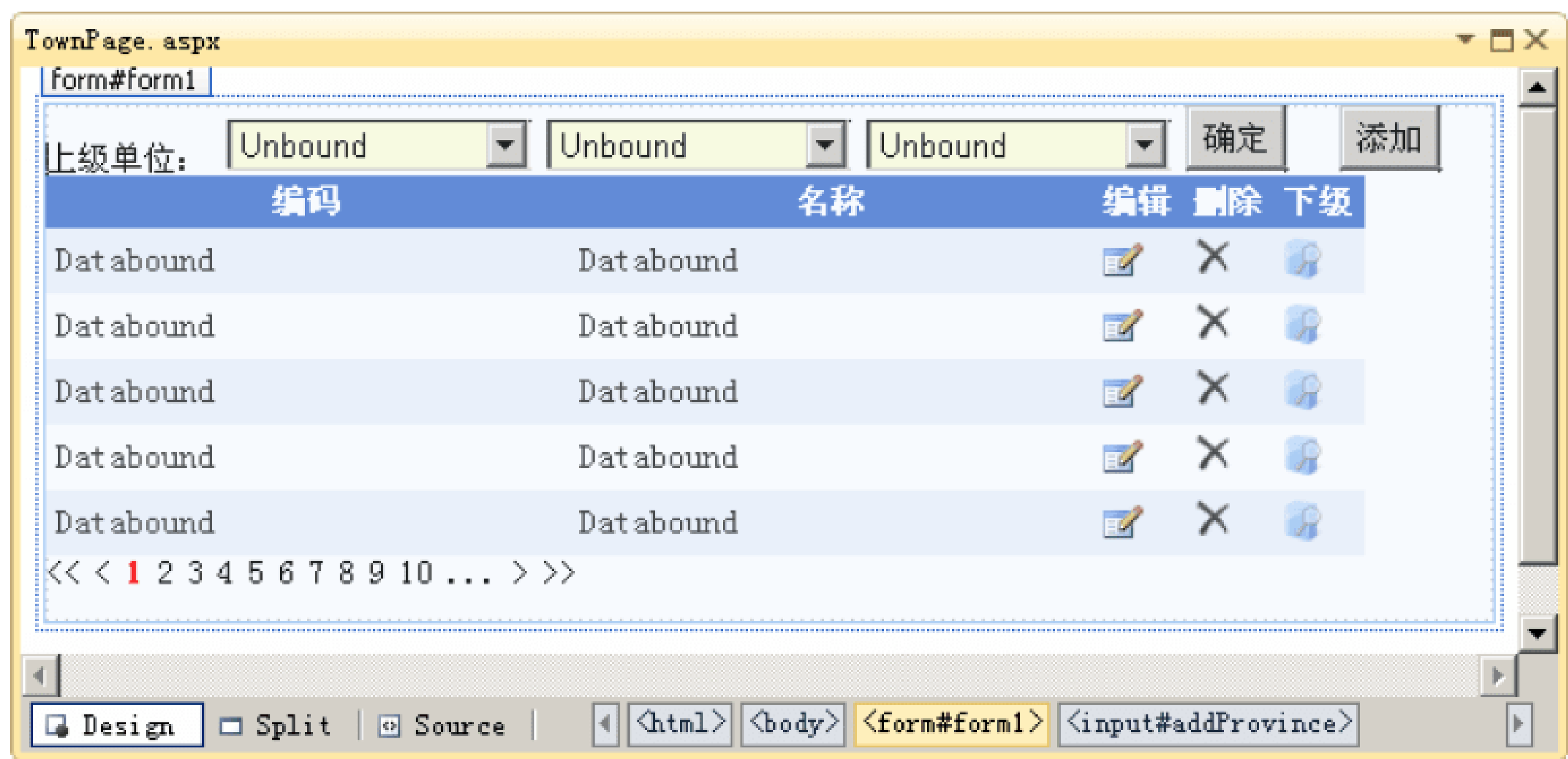


图 14.9　乡镇列表页面

TownPage.aspx 页面代码如下：

```
<form id="form1" runat="server">
 上级单位:
 <uc1:SelectCounty ID="SelectCounty1" runat="server" /><%--选择区县用户控件
 --%>
 <asp:Button ID="filterButton" runat="server" Text="确定" onclick=
 "filterButton Click" />  
 <input id="addProvince" type="button" value="添加" />
 <%--乡镇列表 GridView--%>
 <asp:GridView ID="grid" runat="server" AutoGenerateColumns="false"
    DataKeyNames="ID" onrowcommand="grid RowCommand">
    <Columns>
        <asp:BoundField DataField="ID" HeaderText="编码">
        <HeaderStyle Width="200px" />
        </asp:BoundField>
        <asp:BoundField DataField="Name" HeaderText="名称">
        <HeaderStyle Width="200px" />
        </asp:BoundField>
        <asp:TemplateField HeaderText="编辑">
        <ItemTemplate>
        <img src="../images/edit.png" alt="编辑" onclick="edit(this)"
        style="cursor:pointer;" />
        </ItemTemplate>
        </asp:TemplateField>
         <asp:TemplateField HeaderText="删除">
         <ItemTemplate>
         <asp:ImageButton ID="ImageButton1" ImageUrl="../images/
        delete.png"
                 OnClientClick="return $SjlUtility.confirmDelete()"
                 CommandName="mydelete"
         runat="server" CommandArgument='<%# Eval("ID") %>' />
         </ItemTemplate>
        </asp:TemplateField>
         <asp:TemplateField HeaderText="下级">
         <ItemTemplate>
         <a href="VilligePage.aspx?parent=<%# Eval("ID") %>"> <%--查看下级
         行政（村）--%>
         <img src="../images/search.png" alt="查看" style="border:none;
         height:16px;" /></a>
         </ItemTemplate>
        </asp:TemplateField>
```

```
    </Columns>
</asp:GridView>
<webdiyer:AspNetPager              ID="pager1"              runat="server"
onpagechanged="pager1_PageChanged">
</webdiyer:AspNetPager>
</form>
```

（12）在 TownPage.aspx 页面的 Page_Load 事件中，根据 QueryString 传递过来的区县编号加载乡镇列表。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        initData();
    }
}
private void initData()
{
    pager1.CurrentPageIndex = 1;
    string pid = Request.QueryString["parent"];
                                                  //取得 QueryString 中的区县编号
    //如果 QueryString 不包含区县编号则使用默认区县
    if (string.IsNullOrEmpty(pid))
        pid = ConstValues.DefaultCounty;
    SelectCounty1.selectedCounty = pid;
    loadData(pid);
}
//根据区县编号加载乡镇数据
private void loadData(string parent)
{
    if (string.IsNullOrEmpty(parent) || parent == "-1")
        return;
    TownBLL worker = new TownBLL();
    PageDataArgument arg = new PageDataArgument(pager1.CurrentPageIndex -
    1, pager1.PageSize, true);
    grid.DataSource = worker.getTownsInCounty(parent, arg);
    pager1.RecordCount = arg.count;
    grid.DataBind();
}
```

（13）在 TownPage.aspx 页面的“确定”按钮中，根据所选择的区县重新加载乡镇数据。

```
private void loadData()
{
    loadData(SelectCounty1.selectedCounty);
}
```

（14）在 TownPage.aspx 页面 GridView 控件的 RowCommand 事件中，如果用户选择了删除命令，则删除当前乡镇。

```
protected void grid_RowCommand(object sender, GridViewCommandEventArgs e)
{
    if (e.CommandName == "mydelete")
    {
        TownBLL worker = new TownBLL();
        worker.deleteTown(e.CommandArgument.ToString());
        loadData();
    }
}
```

（15）当在 TownPage.aspx 页面单击“添加”按钮，或者在 GridView 中选择“编辑”命令时，则以模式窗口打开另外一个页面 TownEditPage.aspx，并在其中编辑乡镇数据。“添加”和“编辑”在浏览端通过 JavaScript 实现，代码如下：

```
<script type="text/javascript">
    $(function () {
        //为 GridView 添加光棒效果
        $('#grid').find('tr').hover(
        function () { $(this).addClass("hoverRow"); },
        function () { $(this).removeClass("hoverRow"); });
        //当单击添加按钮时执行 insert 函数
        $('#addProvince').click(insert);
    });          //$(function(){})
    var dialogFeature = "dialogWidth:400px;dialogHeight:300px;";
    //以模式窗口打开 TownEditPage.aspx 进行数据添加
    function insert() {
        if(window.showModalDialog("TownEditPage.aspx",dialogFeature))
        window.location.reload();
    }
    //编辑 GridView 中所选中的乡镇
    //参数 e 为 GridView 中的待编辑行中的"编辑"图片
    function edit(e) {
        var tr = $(e).closest('tr');            //找到待编辑行
        var id = tr.find('td').eq(0).html();
         //待编辑行的第一个单元格内容即为乡镇编号
        //以模式窗口打开新页面，然后重新加载数据
        if(window.showModalDialog("TownEditPage.aspx?id=" + id,
        dialogFeature))
            window.location.reload();
    }
</script>
```

（16）在表现层项目 Nrcm.Web 中添加一个页面 TownEditPage.aspx，页面功能为编辑和添加乡镇。页面代码如下：

```
<head runat="server">
    <script type="text/javascript">
        $(function () {
            //单击"取消"按钮时关闭对话框，并返回 False
            $('#cancel').click(function(){
            window.returnValue=false; window.close();
            });
        });//$(function)
    </script>
</head>
<body>
    <form id="form1" runat="server">
     <div id="editDialog" style="text-align:center;">
    编码：<asp:TextBox ID="cityCode" runat="server"></asp:TextBox>
         
    名称：<asp:TextBox ID="cityName" runat="server" ></asp:TextBox>
        <br />
        上级单位：<uc1:SelectCounty ID="county" runat="server" />
        <br /><br />
        <asp:Button ID="OK" runat="server" Text="保存" onclick="OK_Click" />
         
        <input type="button" id="cancel" value="取消"/>
```

```
    </div>
    </form>
</body>
public partial class TownEditPage : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            string oldid = Request.QueryString["id"] ?? "";
                                    //得到所编辑的乡镇 ID
            if (oldid.Length == 0) oldid = ConstValues.DefaultTown;
            if (oldid.Length > 0)
            {
                //在页面控件上显示乡镇信息
                var theTown = new TownBLL().getTownById(oldid);
                county.selectedCounty = theTown.ParentID;
                cityCode.Text = theTown.ID;
                cityName.Text = theTown.Name;
            }
        }
    }
    //保存被编辑（或添加）的乡镇
    private void save()
    {
        string oldid=Request.QueryString["id"]??"";
        Town v = Town.CreateTown(cityCode.Text.Trim(),
            cityName.Text.Trim(), county.selectedCounty);
        TownBLL worker = new TownBLL();
        if (oldid.Length > 0)
        {
            //主键未改变，则修改原有乡镇数据
            if (oldid == cityCode.Text.Trim())
            {
                worker.updateTown(v);
                return;
            }
            worker.deleteTown(v.ID);        //主键改变，先删除再添加
        }

        worker.addTown(v);                  //添加乡镇
    }

    protected void OK_Click(object sender, EventArgs e)
    {
        save();
        //通过 JavaScript 关闭对话框并返回 true
        this.ClientScript.RegisterStartupScript(this.GetType(),
            "closeDialog",
"<script>window.returnValue=true;window.close();</script>");
    }
}
```

14.3.3　分段报销比例

新农合采用分段报销的方式，参合农民住院费用在不同的区间段享受不同的报销比例，如 500 元以下不报销（报销比例为 0），500～2000 元报销 50%，2000～5000 报销 55%，

依此类推。分段报销比例中的分界点和各段报销比例都是可以设置的。分段报销比例保存在数据库的 CompensateRangeRate 表中，此表共有以下 4 个字段。

- ID：int 类型，主键，标识列。
- Low：int 类型，分段起点。
- High：int 类型，分段终点。
- Rate：float 类型，报销比例。

（1）在实体框架层 Nrcm.Entity 项目中，添加对应 CompensateRangeRate 表的实体模型。

（2）在数据访问层 Nrcm.Dal 项目中添加一个类 CompensateRateDal，编写相关数据访问代码。

```
public static class CompensateRateDAL
{
    public static int add(Rate rate)
    {
        return EntityUtility.add<MedicineContext, Rate>(rate);
    }
    public static int update(Rate rate)
    {
        return EntityUtility.update<MedicineContext, Rate>(rate);
    }
    public static List<Rate> getAll()
    {
       return EntityUtility.selectMany(new MedicineContext().
       CompensateRangeRate, r => r.Low, PageDataArgument.allData);
    }
    public static int deleteByID(int id)
    {
       return EntityUtility.delete(new MedicineContext().Compensate
       RangeRate, r => r.ID == id);
    }
}
```

（3）在业务逻辑层项目 Nrcm.Bll 中添加一个类 CompensateRateBll，这个类的功能主要是调用数据访问层 CompensateDal 类的相应方法，此处省略类的代码。

（4）在表现层项目 Nrcm.Web 中添加一个页面 CompensateRatePage.aspx，页面布局如图 14.10 所示。

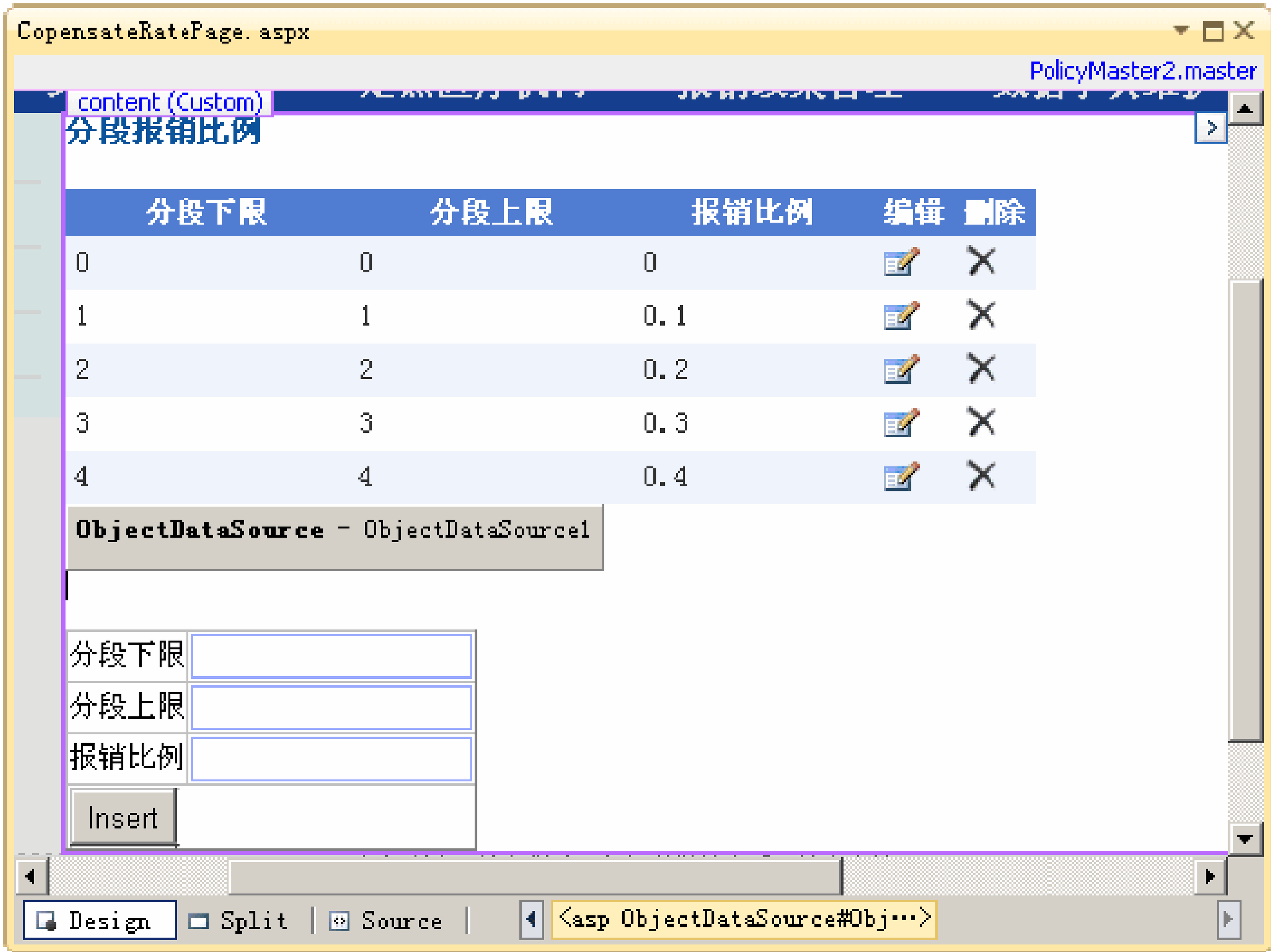


图 14.10　分段报销比例设置页面

CompensateRatePage.aspx 页面上部使用 GridView 显示目前已有的分段报销数据，下部使用 DetailsView 进行数据添加，使用 ObjectDataSource 调用业务逻辑层 CompensateRateBll 类实现数据增删改查。页面代码如下：

```
<asp:Content ID="Content2" ContentPlaceHolderID="content" runat="server">
<h3>分段报销比例</h3>
   <%--分段报销比例数据列表--%>
   <asp:GridView ID="grid1" runat="server" AutoGenerateColumns="False"
     CssClass="grid" DataKeyNames="ID"DataSourceID="ObjectDataSource1" >
      <Columns>
         <asp:BoundField DataField="ID" HeaderText="ID" ReadOnly="True"
         Visible="false"  InsertVisible="False" SortExpression="ID"  />
         <asp:BoundField DataField="Low" HeaderText="分段下限" ItemStyle-
         Width="120"  >
         </asp:BoundField>
         <asp:BoundField DataField="High" HeaderText="分段上限"
         ItemStyle-Width="120"  >
         </asp:BoundField>
         <asp:BoundField DataField="Rate" HeaderText="报销比例"
         ItemStyle-Width="100"/>
          <asp:CommandField ButtonType="Image" ShowEditButton="true"
         EditImageUrl="../images/edit.png" HeaderText="编辑" />
          <asp:CommandField ButtonType="Image" ShowDeleteButton="true"
         DeleteImageUrl="../images/delete.png" HeaderText="删除" />
      </Columns>
   </asp:GridView>
   <asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
      DataObjectTypeName="Nrcm.Entity.CompensateRangeRate"
      DeleteMethod="delete"
      InsertMethod="add" OldValuesParameterFormatString="original_{0}"
      SelectMethod="getAll" TypeName="Nrcm.Bll.Medicine.
      CompensateRateBLL"
      UpdateMethod="update">
   </asp:ObjectDataSource>
   <br />    <br />
   <%--DetailsView 用于添加新的分段数据--%>
   <asp:DetailsView ID="DetailsView1" runat="server" AutoGenerateRows=
   "False"
      DataKeyNames="ID" DataSourceID="ObjectDataSource1" DefaultMode=
      "Insert"  >
      <Fields>
         <asp:BoundField DataField="ID" HeaderText="ID" InsertVisible=
         "False"
            ReadOnly="True" SortExpression="ID" />
         <asp:BoundField DataField="Low" HeaderText="分段下限" />
         <asp:BoundField DataField="High" HeaderText="分段上限" />
         <asp:BoundField DataField="Rate" HeaderText="报销比例"  />
         <asp:CommandField ButtonType="Button" ShowInsertButton="True"
         ShowCancelButton="false" />
      </Fields>
   </asp:DetailsView>
</asp:Content>
```

（5）新农合采用分段报销制度，即对于一个住院消费金额，并不是乘以一个报销比例就得到报销金额，而是要将消费总额划分成各个区间，每个区间享受不同的报销比例，然后将各个区间的报销金额相加，就得到总的报销金额。举例来说，分段报销比例为 500 元

以下不报销（报销比例为 0），500–2000 元报销 50%，2000–5000 报销 55%，5000–10000 报销 60%，10000 以上报销 70%。那么对于 15000 元的住院费用报销金额为：

（500–0）×0+（2000–500)×0.5+（5000–2000）×0.55+（10000–5000）×0.6=5400

在新农合系统中，使用数据库中的自定义函数实现上述报销金额计算功能，代码如下：

```
CREATE FUNCTION [dbo].[CalculateCompensation]
--根据分段报销比例计算应报销金额
(@totalmoney float)                                   --参数：总金额
RETURNS float                                         --返回报销金额
AS
BEGIN
if @totalmoney is null return null;
if @totalmoney=0 return 0;
declare @remain float;
declare @compensation float,@temp float;
declare @low float,@rate float;
set @low=-1;
--确定费用所属的最高报销级别
select @low=low from CompensateRangeRate
where low<=@totalmoney and high>=@totalMoney;
if @low<0 return 0;
select @rate=rate from CompensateRangeRate where low=@low;
set @compensation=(@totalmoney-@low+1)*@rate;
--得到最低级别至次高级别应得的报销金额
select @temp=sum((high-low)*rate) from CompensateRangeRate
where low<@low;
set @compensation=@compensation+@temp
return @compensation
END
```

14.4　家庭档案管理

新农合参合和缴费通常以家庭为单位，在新农合系统中需要对家庭和农民信息进行查询和维护。

14.4.1　数据库表和实体类

与农民档案相关的有两个表：家庭表 FamilyInfo 和人员表 PersonInfo。两个表结构如下：

```
CREATE TABLE [dbo].[FamilyInfo](
    [ID] [nvarchar](20) NOT NULL,                     --家庭编号
    [Name] [nvarchar](10) NOT NULL,                   --户名
    [Village] [nvarchar](50) NOT NULL,                --所属村庄
    [Address] [nvarchar](50)  NULL,                   --地址
    [Post] [nchar](6)  NULL,                          --邮编
    [Phone] [nvarchar](20)  NULL,                     --联系电话
    [CMAttribute] [nchar](2) NULL ,                   --参合属性
    [FamilyAttribute] [nchar](2) NULL ,               --户属性
 CONSTRAINT [PK_FamilyInfo] PRIMARY KEY CLUSTERED
```

```
(    [ID] ASC     )
CREATE TABLE [dbo].[PersonInfo](
    [ID] [nvarchar](20)  NOT NULL,                        --个人编号（身份证号）
    [Name] [nvarchar](10)  NOT NULL,                      --姓名
    [Sex] [nchar](1)  NOT NULL,                           --性别
    [Birth] [datetime] NULL,                              --出生日期
    [Nationality] [nchar](2)  NULL,                       --民族
    [Marriage] [nchar](2)  NULL,                          --婚姻状况
    [Health] [nchar](2)  NULL,                            --健康状况
    [MedicalCard] [nvarchar](20)  NULL,                   --参合卡号
    [Occupation] [nchar](2)  NULL,                        --职业
    [WorkAddress] [nvarchar](50)  NULL,                   --工作单位
    [Phone] [nvarchar](20)  NULL,                         --联系电话
    [FamilyID] [nvarchar](20)  NOT NULL,                  --所属家庭编号
    [FamilyRelation] [nchar](2)  NULL,                    --与户主关系
    [Village] [nvarchar](10)  NULL,                       --所属村庄
    [CMAttribute] [nchar](2)  NULL,                       --参合属性
 CONSTRAINT [PK PersonInfo] PRIMARY KEY CLUSTERED
(    [ID] ASC     )
```

在 FamilyInfo 表中有一个 Name 字段，现实生活中并没有家庭名称这个说法，但是在新农合系统中，当显示家庭列表时，为了直观地表示一个家庭，需要给这个家庭一个唯一且可读的属性，显然家庭编号不能起到这个作用。因为家庭编号全是字母数字，不具有可读性，所以给 FamilyInfo 表中添加一个 Name 字段，以直观地描述家庭信息，这个字段的值通常为“张三家”、“李四户”之类。在实体框架层 Nrcm.Entity 项目中添加与以上两表对应的数据模型，如图 14.11 所示。

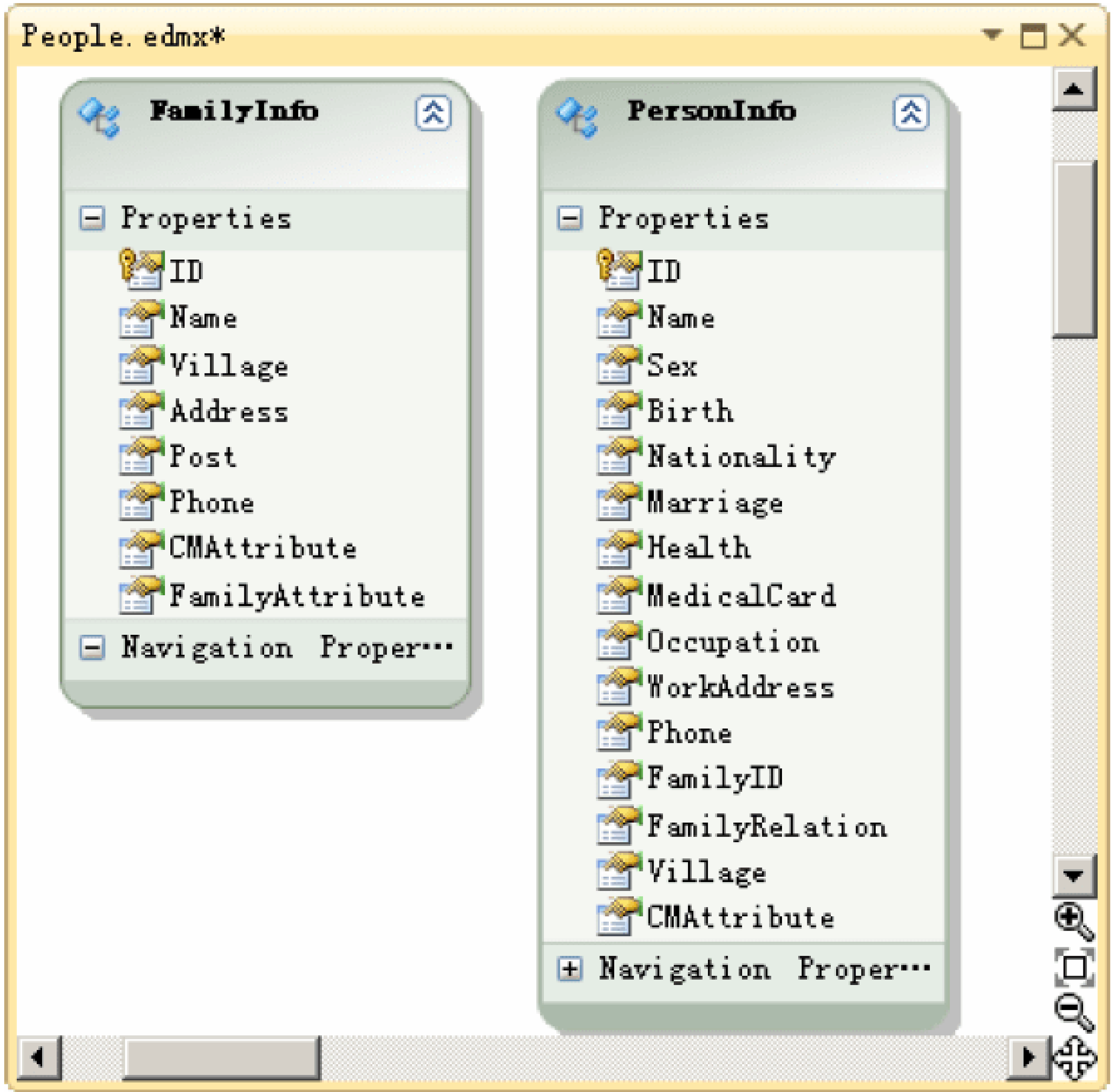


图 14.11　家庭和个人实体模型

在数据库中的 FamilyInfo 和 PersonInfo 表中，职业、民族、健康状态、参合属性等字段都是外键，其中保存的是编码，在实体类中这些属性的值也是编码。在页面上显示这些数据时，需要显示为可读的文本，而不是编码。为了便于将数据绑定到 GridView 等数据控

件，在自动生成的实体类中，为职业、民族等每一个编码添加一个属性，表示此编码所对应的文本。

修改实体模型类时，要注意不要修改自动生成的文件，因为这些自动生成的文件随时可能被刷新，在其中的修改会被覆盖。Visual Studio 自动生成的实体类是一个分部类（Partial Class），允许把一个类的代码写到多个文件中。如果要对自动生成的实体类进行修改，应该添加一个新的文件，并把需要添加的属性或方法写到新的文件中。对 FamilyInfo 和 PersonInfo 类的扩展代码如下：

```
public partial class FamilyInfo
{
    public string VillageName { get; set; }                //村庄名称
    public string CMAttributeText { get; set; }            //参合属性文本
    public string FamilyAttributeText { get; set; }        //家庭属性文本
}
public partial class PersonInfo
{
    public string villageName { get; set; }                //村庄名称
    public string familyName { get; set; }                 //户名
    public string relationName { get; set; }               //与户主关系
    public string nationalityName { get; set; }            //民族名称
    public string marriageState { get; set; }              //婚姻状况
    public string healthText { get; set; }                 //健康状况
    public string occupationName { get; set; }             //职业名称
    public string cmAttributeText { get; set; }            //参合属性文本
}
```

14.4.2　家庭信息管理

在新农合系统中，可以浏览某一村的所有家庭信息，编辑、删除这些家庭信息，查看家庭成员列表等。

（1）在数据访问层项目 Nrcm.Dal 中添加一个 FamilyDAL 类，实现与家庭信息相关的数据访问功能。

```
public static class FamilyDAL
{
    #region public methods
    //添加家庭信息
    public static int addFamily(FamilyInfo family)
    {
        return EntityUtility.add<PeopleContext, FamilyInfo>(family);
    }
    //根据家庭 ID 删除家庭
    public static int deleteByID(string id)
    {
        return EntityUtility.delete( getQuery(), f => f.ID == id);
    }
    //修改家庭信息
    public static int updateFamily(FamilyInfo family)
    {
        return EntityUtility.update<PeopleContext, FamilyInfo>(family);
    }
```

```
    //根据 ID 得到家庭信息
    public static FamilyInfo getByID(string id)
    {
        var family= EntityUtility.selectOne(getContext().FamilyInfo, f =>
        f.ID == id);
        loadDetail(family);
        return family;
    }
    //得到某村庄的所有家庭
    public static List<FamilyInfo> getFamiliesInVillage(string village,
    PageDataArgument pageArg)
    {
        using (var context = getContext())
        {
            var list=EntityUtility.selectMany(context.FamilyInfo, f => f.
            ID, pageArg, f => f.Village == village);
            list.ForEach(f => loadDetail(f));
            return list;
        }
    }
    #endregion
    #region private methods
    //得到 ObjectQuery 对象
    private static ObjectQuery<FamilyInfo> getQuery()
    {
        return new PeopleContext().FamilyInfo;
    }
    //得到 DataContext 对象
    private static PeopleContext getContext()
    {
        return new PeopleContext();
    }
    //加载家庭详细信息（外键数据）
    private static void loadDetail(FamilyInfo family)
    {
        if (family == null)
            return;
        //得到家庭的参合属性
        if(family.CMAttribute!=null)
            family.CMAttributeText = SmallDictionaryDAL.getByID
                (SmallDictionaryTables.TableCMAttribute, family.
                CMAttribute).name;
        //得到家庭属性
        if(family.FamilyAttribute!=null)
            family.FamilyAttributeText = SmallDictionaryDAL.getByID
                (SmallDictionaryTables.TableFamilyAttribute, family.
                FamilyAttribute).name;
        //得到村庄名称
        if(family.Village!=null)
            family.VillageName = new VillageDAL().getVillageById(family.
            Village).Name;
    }
    #endregion
}
```

（2）在业务逻辑层 Nrcm.Bll 项目中添加一个 FamilyBLL 类。FamilyBLL 类主要功能为调用 FamilyDAL 类的相应方法，此处省略其代码。

（3）在表现层项目 Nrcm.Web 中添加一个页面 FamilyPage.aspx，页面布局如图 14.12

所示。

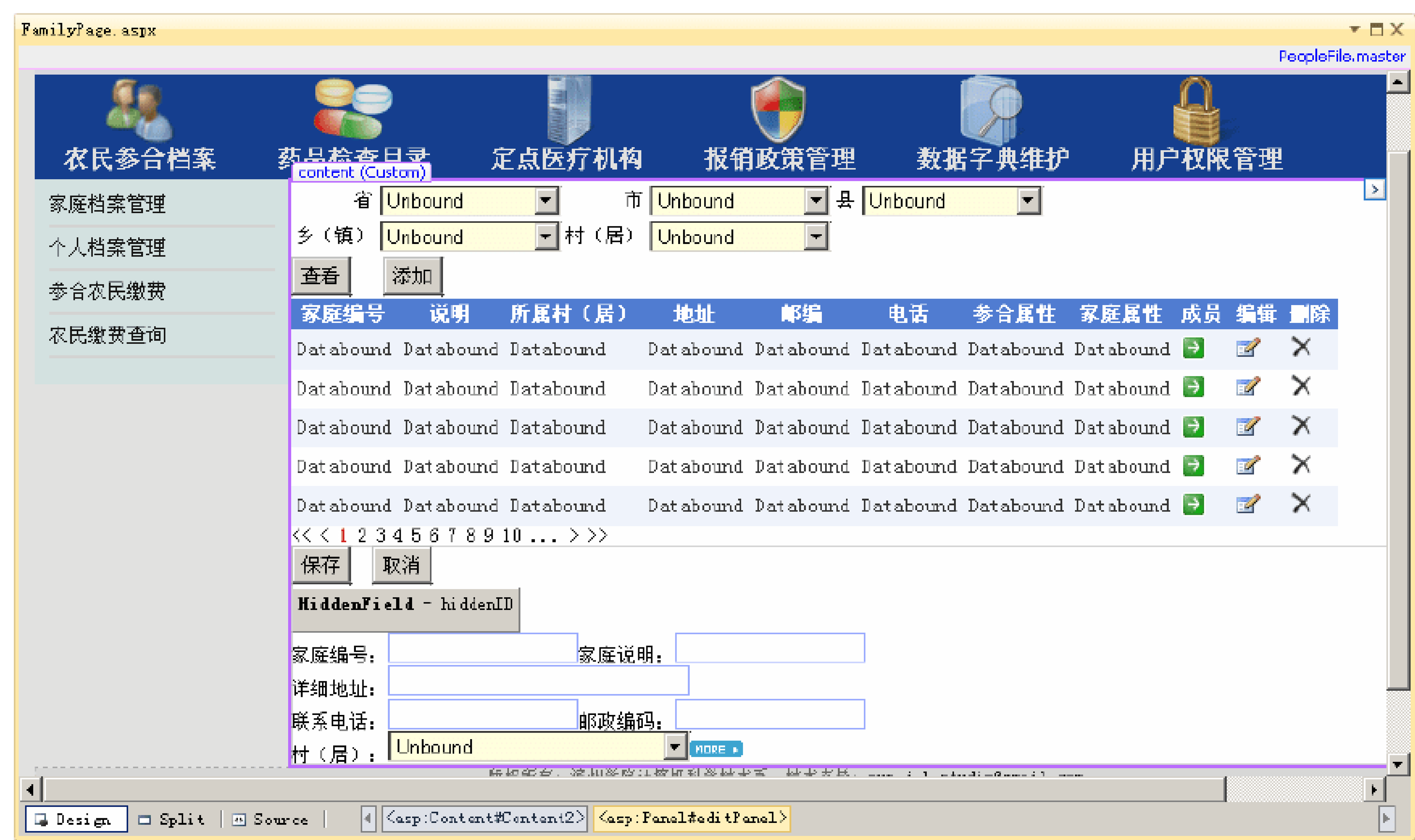


图 14.12　家庭列表页面布局

FamilyPage.aspx 页面可以查看某一村庄的所有家庭列表。页面最顶部为一个用户控件，此用户控件包含一组级联下拉列表，可以选择村庄。用户控件 SelectVillage.ascx 代码如下：

```
<table style="text-align:right;">
<tr>
<td>省</td>
<td><asp:DropDownList ID="province" runat="server" AutoPostBack="true"
    DataValueField="ID" DataTextField="Name"
    onselectedindexchanged="province_SelectedIndexChanged">
</asp:DropDownList></td>
<td>市</td>
<td><asp:DropDownList ID="city" runat="server" AutoPostBack="true"
    DataValueField="ID" DataTextField="Name"
    onselectedindexchanged="city_SelectedIndexChanged">
</asp:DropDownList></td>
<td>县</td><td><asp:DropDownList ID="county" runat="server"
    DataValueField="ID" DataTextField="Name" AutoPostBack="true"
    onselectedindexchanged="county_SelectedIndexChanged" >
</asp:DropDownList></td>
</tr>
<tr>
<td>乡（镇）</td>
<td><asp:DropDownList ID="town" runat="server" AutoPostBack="true"
    DataValueField="ID" DataTextField="Name"
    onselectedindexchanged="town_SelectedIndexChanged">
</asp:DropDownList></td>
<td>村（居）</td>
<td><asp:DropDownList  ID="village"  runat="server"   DataValueField="ID"
DataTextField="Name" >
</asp:DropDownList></td>
<td></td><td></td>
</tr>
</table>
```

在图 14.12 所示的 FamilyPage.aspx 页面中选中村庄后，单击“确定”按钮，即可在页面中部的 GridView 中显示此村所有户。GridView 最右侧三列为命令列，单击相应命令可以查看此户成员列表、编辑此户信息、删除此户。页面底部为编辑和添加户的区域，用户输入数据，单击“保存”按钮，则将此户信息保存到数据库。FamilyPage.aspx 页面代码如下：

```
<asp:Content ID="Content2" ContentPlaceHolderID="content" runat="server">
    <uc1:SelectVillage ID="SelectVillage1" runat="server" />
    <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="
    查看" />
      <asp:Button  ID="addButton" runat="server" Text="添加"
    onclick="addButton_Click" />
    <%--用于显示家庭列表的 GridView--%>
    <asp:GridView ID="grid1" runat="server" AutoGenerateColumns="False"
    OnRowCommand="GridView1_RowCommand" DataKeyNames="ID"
    CssClass="grid">
        <Columns>
            <asp:BoundField DataField="ID" HeaderText="家庭编号" />
            <asp:BoundField DataField="Name" HeaderText="说明" />
            <asp:BoundField DataField="VillageName"  HeaderText="所属村（居）
            "/>
            <asp:BoundField DataField="Address" HeaderText="地址"  />
            <asp:BoundField DataField="Post" HeaderText="邮编" />
            <asp:BoundField DataField="Phone" HeaderText="电话"  />
            <asp:BoundField DataField="CMAttributeText" HeaderText="参合属性
            " />
            <asp:BoundField DataField="FamilyAttribute" HeaderText="家庭属性
            " />
             <asp:BoundField DataField="Village" Visible="false" />
             <asp:BoundField DataField="CMAttribute" Visible="false" />
            <asp:BoundField DataField="FamilyAttribute" Visible="false" />
            <%--以下列为模板列，包含一个图片链接，链接到另一个页面显示家庭成员--%>
            <asp:TemplateField HeaderText="成员">
             <ItemTemplate>
             <a href="../people/FamilyMember.aspx?family=<%#Eval("ID")%>">
             <img  style="border:none;" src="../images/right2.png" alt="家
            庭成员"/></a>
             </ItemTemplate>
             </asp:TemplateField>
             <%--模板列，包含一个图片按钮，单击此按钮则编辑当前家庭，通过 JavaScript
            实现--%>
             <asp:TemplateField HeaderText="编辑">
             <ItemTemplate>
             <asp:ImageButton ID="editButton" runat="server" ImageUrl=
            "../images/edit.png" CommandName="edit0" CommandArgument=
            '<%#Eval("ID")%>'  />
             </ItemTemplate>
             </asp:TemplateField>
             <%--模板列，包含一个图片按钮，单击此按钮则提示确认后删除当前家庭--%>
             <asp:TemplateField HeaderText="删除">
             <ItemTemplate>
             <asp:ImageButton ID="deleteButton" runat="server" ImageUrl=
            "../images/delete.png" CommandName="delete0"   CommandArgument=
            '<%#Eval("ID")%>' OnClientClick="return deleteConfirm();"  />
             </ItemTemplate>
             </asp:TemplateField>
```

```
        </Columns>
    </asp:GridView>
    <%--分页控件--%>
     <webdiyer:AspNetPager
        ID="pager" runat="server" onpagechanged=
        "AspNetPager1_PageChanged">
    </webdiyer:AspNetPager>
    <%--以下为编辑区域，可以编辑和添加家庭信息--%>
   <asp:Panel ID="editPanel" runat="server" Visible="false">
       <asp:Button ID="saveButton" runat="server" Text="保存"
           onclick="saveButton_Click" /> 
       <asp:Button ID="cancelButton" runat="server" onclick=
       "cancelButton_Click"
           Text="取消" /> <br />
       <asp:HiddenField ID="hiddenID" runat="server" />
       家庭编号：<asp:TextBox ID="familyID" runat="server"></asp:TextBox>
       家庭说明：<asp:TextBox ID="descriptoin" runat="server"></asp:TextBox>
       <br />
       详细地址：<asp:TextBox ID="address" runat="server" width=
       "200px"></asp:TextBox><br />
       联系电话：<asp:TextBox ID="phone" runat="server"></asp:TextBox>
       邮政编码：<asp:TextBox ID="post" runat="server"></asp:TextBox> <br />
       村（居）：<asp:DropDownList ID="villageList" runat="server"
       DataTextField="Name"
           DataValueField="id" Width="200">
       </asp:DropDownList>
       <%--单击以下图片按钮则以模式窗口打开另一页面，在其中可选择村庄--%>
       <img ID="moreVillage" alt="更多村（居）" src="../images/more.gif"
           style="cursor:pointer;" />   <br />
   </asp:Panel>
</asp:Content>
```

（4）在 FamilyPage.aspx 页面中编写 JavaScript 代码，当单击“删除”按钮时给出确认提示，单击“选择村庄”图片按钮时打开模式对话框选择村庄。

```
<script type="text/javascript">
    $(function() {
        $("#moreVillage").click(selectVillage);
    });
    var villageID = '#<%=villageList.ClientID %>'; //村庄列表控件 ID
//删除确认
function deleteConfirm() {
    return confirm("确实要删除吗？");
}
//打开模式对话框，选择家庭所属村庄
function selectVillage() {
    var village = $('#ctl00_ctl00_content_content_SelectVillage1_
    village')[0].value;
    if (!village) village = "";
    var a = window.showModalDialog("../Dialog/SelectVillagePage.aspx?
    village=" + village,
        null, "dialogWidth=500px;dialogHeight=200px;");
    if (!a)                                 //如果对话框没有返回结果函数返回
        return;
    //对话框的返回结果形式为村庄 ID，村庄名称
    //将返回结果从逗号（,）处拆开，可得到村庄 ID 和名称，将其显示在下拉列表中
    var s = a.split(',');
    var o = "<option value='" + s[0] + "'>" + s[1] + "</option>";
```

```
    $(villageID).append(o);
    $(villageID).val(s[0]);
}
</script>
```

（5）在 FamilyPage.aspx 页面上“查看”按钮的 Click 事件中，根据所选择的村庄，读取家庭数据并绑定到 GridView 控件上。

```
protected void Button1_Click(object sender, EventArgs e)
{
    pager.CurrentPageIndex = 1;
    editPanel.Visible = false;
    bindGrid();
    bindVillageList();
}
//绑定编辑区域的村庄列表
private void bindVillageList()
{
    villageList.Items.Clear();
    string id = SelectVillage1.selectedVillage;
    Village v = new VillageBLL().getVillageById(id);   //得到所选择的村庄
    if (v == null) return;
    PageDataArgument arg = PageDataArgument.allData;
    var list = new VillageBLL().getVillagesInTown(v.ParentID, arg);
                                                  //得到同一乡镇的所有村
    villageList.DataSource = list;
    villageList.DataBind();
}
//绑定家庭列表
private void bindGrid()
{
    grid1.DataSource = null;
    grid1.DataBind();
    string id = SelectVillage1.selectedVillage;
    if (string.IsNullOrEmpty(id)) return;
    PageDataArgument arg = new PageDataArgument(pager.CurrentPageIndex - 1,
    pager.PageSize, true);
    pager.RecordCount = arg.count;
    var list = FamilyBLL.getFamiliesInVillage(id, arg);
    grid1.DataSource = list;
    grid1.DataBind();
}
```

（6）在 FamilyPage.aspx 页面上 GridView 控件的 RowCommand 事件中，处理编辑和删除命令。

```
//GridView 命令处理程序
protected void GridView1_RowCommand(object sender, GridViewCommand
EventArgs e)
{
    //本页面 GridView 支持两种自定义命令：编辑 edit0 和删除 delete0
    //如果是编辑命令，则切换到编辑模式
    if (e.CommandName == "edit0")
    {
        FamilyInfo family = FamilyBLL.getByID(e.CommandArgument.
        ToString());
        if (family == null) return;
        //将待编辑家庭信息显示在编辑区域的控件上
```

```
        familyID.Text = family.ID;
        descriptoin.Text = family.Name;
        phone.Text = family.Phone;
        post.Text = family.Post;
        villageList.SelectedValue = family.Village;
        editPanel.Visible = true;
    }
    //如果是删除命令，则删除指定家庭
    else if (e.CommandName == "delete0")
    {
        FamilyBLL.deleteByID(e.CommandArgument.ToString());
        bindGrid();
    }
}
```

（7）在 FamilyPage.aspx 页面上“添加”按钮的 Click 事件中，清空并显示编辑区域。

```
protected void addButton Click(object sender, EventArgs e)
{
    editPanel.Visible = true;
    clearEdit();
}
//清除编辑区域内容
private void clearEdit()
{
    hiddenID.Value = "-1";
    villageList.SelectedIndex = 0;
    familyID.Text = "";
    descriptoin.Text = "";
    address.Text = "";
    phone.Text = "";
    post.Text = "";
}
```

（8）在 FamilyPage.aspx 页面上“保存”按钮的 Click 事件中，保存当前编辑的家庭信息。要注意区分当前编辑的家庭是新增家庭还是原有家庭，相应调用 Insert 和 Update 方法。

```
//保存正在编辑的家庭信息
protected void saveButton Click(object sender, EventArgs e)
{
    //根据编辑区域各个控件的值构建一个 FamilyInfo 对象
    FamilyInfo family = new FamilyInfo();
    family.ID = familyID.Text;
    family.Name = descriptoin.Text;
    family.Phone = phone.Text;
    family.Post = post.Text;
    family.Village = villageList.SelectedValue;
    //如果正在编辑的家庭 ID 为-1 则说明为新增家庭，否则为编辑已有家庭
    if (hiddenID.Value == "-1")
        FamilyBLL.addFamily(family);
    else
        FamilyBLL.updateFamily(family);
    editPanel.Visible = false;
    clearEdit();
    bindGrid();
}
```

（9）运行 FamilyPage.aspx，运行界面如图 14.13 所示。

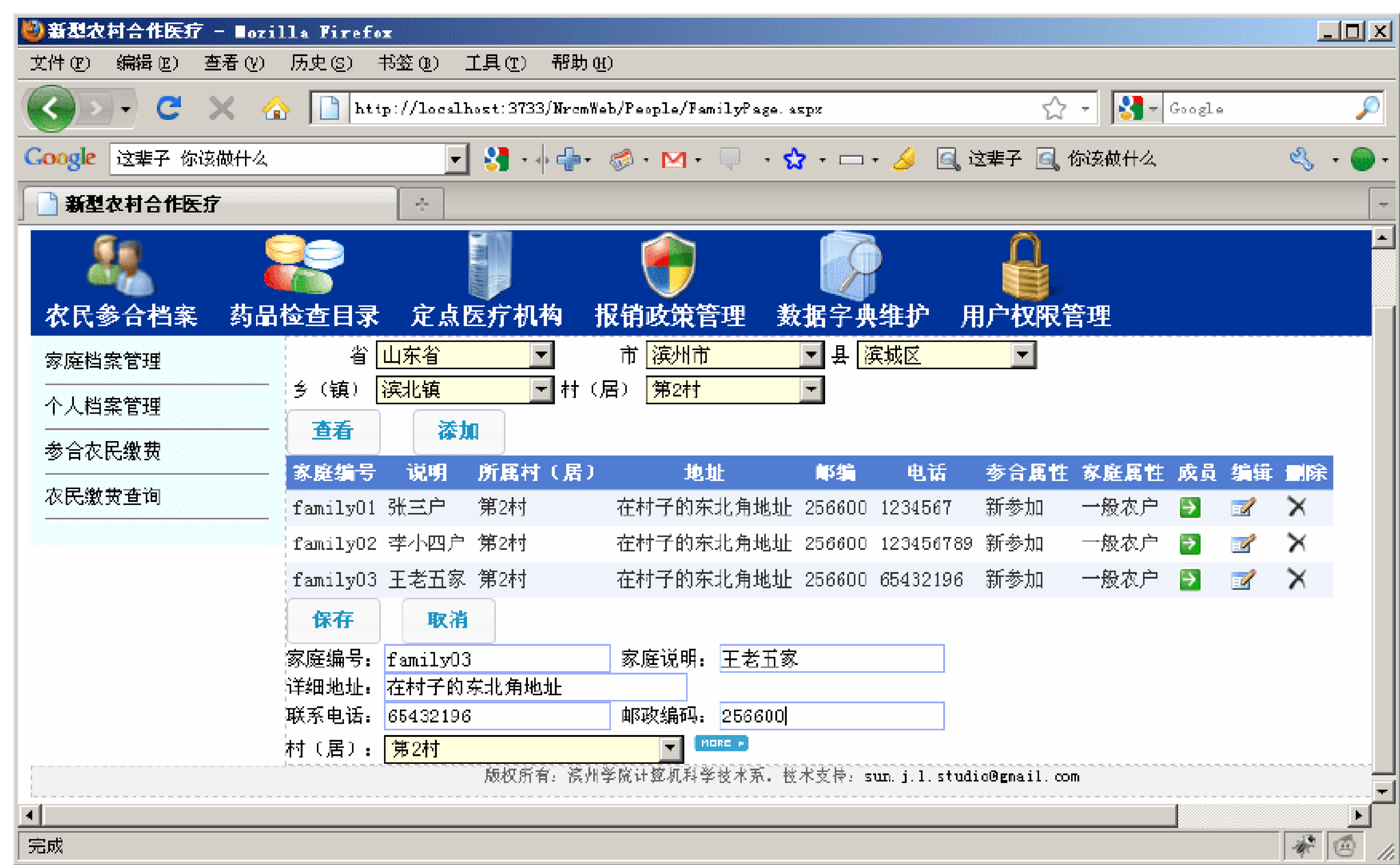


图 14.13　家庭管理页面

14.4.3　参合农民缴费

参加新型农村合作医疗的农民必须按照国家规定的数额每年缴纳一次费用。新农合管理系统中提供缴费功能和缴费查询功能。参合农民缴费数据保存在数据库的 PersonPayment 表中，表结构如下：

```
CREATE TABLE [dbo].[PersonPayment](
    [RecordID] [bigint] IDENTITY(1,1) NOT NULL,    --缴费流水号，主键
    [PersonID] [nvarchar](20) NOT NULL,            --个人编号
    [PayDate] [datetime] NOT NULL DEFAULT (getdate()), --缴费日期
    [Year] [int] NOT NULL,                             --缴费年度
    [PayMoney] [money] NOT NULL,                       --缴费金额
 CONSTRAINT [PK_PersonPayment] PRIMARY KEY CLUSTERED
(    [RecordID] ASC  )
```

当农民缴费时，需要向 PersonPayment 表中添加一条数据，其中缴费年度和缴费金额都保存在数据库的 CompensateSpecialPolicy 表中。参合农民缴费功能由数据库的存储过程实现，代码如下：

```
ALTER PROCEDURE [dbo].[sp_PersonPay]
@personid nvarchar(20)                          --个人编号
AS
BEGIN
set nocount on;
declare @amount money;
declare @year int;
--得到缴费金额和参合年度
select @amount=[Value] from CompensateSpecialPolicy where [ID]='05';
select @year=[Value] from  CompensateSpecialPolicy where [ID]='04';
insert into PersonPayment (PersonID,Year,PayMoney)
values (@personid,@year,@amount);
```

```
set @amount=0
delete from PersonAccount where PersonId=@personid and Year=@year;
declare @balance1 money,@balance2 money
--缴费后，重新设置年度可报余额
select  @balance1=[Value] from CompensateSpecialPolicy where [ID]='01';
select  @balance2=[Value] from CompensateSpecialPolicy where [ID]='03';
insert into PersonAccount(PersonID,Year,HospitalBalance,ClinicBalance)
values (@personid,@year,@balance1,@balance2);
--设置参合属性为连续参合
update PersonInfo set cmAttribute ='03' where id=@personid;
END
```

（1）在数据访问层项目 Nrcm.Dal 中添加一个类 PersonPaymentDAL，代码如下：

```
public static class PersonPaymentDAL
{
    //参合农民缴费，调用存储过程实现
    public static void pay(string personid)
    {
        Database db = DatabaseFactory.CreateDatabase("NRCM");
        db.ExecuteNonQuery("sp_personpay",personid);
    }
    //得到某人某年的缴费信息
    public static PersonPayment getByPersonAndYear(string person, int year)
    {
        return EntityUtility.selectOne(getQuery(), p => p.PersonID ==
        person && p.Year == year);
    }
    //得到某人所有缴费情况
    public static List<PersonPayment> getByPerson(string person)
    {
        return EntityUtility.selectMany(getQuery(), p => p.Year,
            Nrcm.Common.PageDataArgument.allData, p => p.PersonID ==
            person);
    }
    private static System.Data.Objects.ObjectQuery<PersonPayment>
    getQuery()
    {
        return new ExpenseContext().PersonPayment;
    }
}
```

（2）在业务逻辑层项目 Nrcm.Bll 中添加一个 PersonPaymentBLL 类，其主要功能为调用数据访问层 PersonPaymentDAL 的相应功能，此处省略其代码。

（3）在表现层项目 Nrcm.Web 中添加一个缴费页面 PersonPayPage.aspx，页面布局如图 14.14 所示。

在图 14.14 所示的参合缴费页面中，输入身份证号，单击“查看”按钮，可以查看此人的详细信息，以避免由于输入错误编号而误给他人缴费。页面下部显示个人信息的区域为一个用户控件 PersonControl，受篇幅限制，此处不给出该控件的实现过程，其详细代码可参见本书配套光盘。在 PersonPayPage.aspx 页面中单击“缴费”按钮即可完成缴费。PersonPayPage.aspx 页面代码如下：

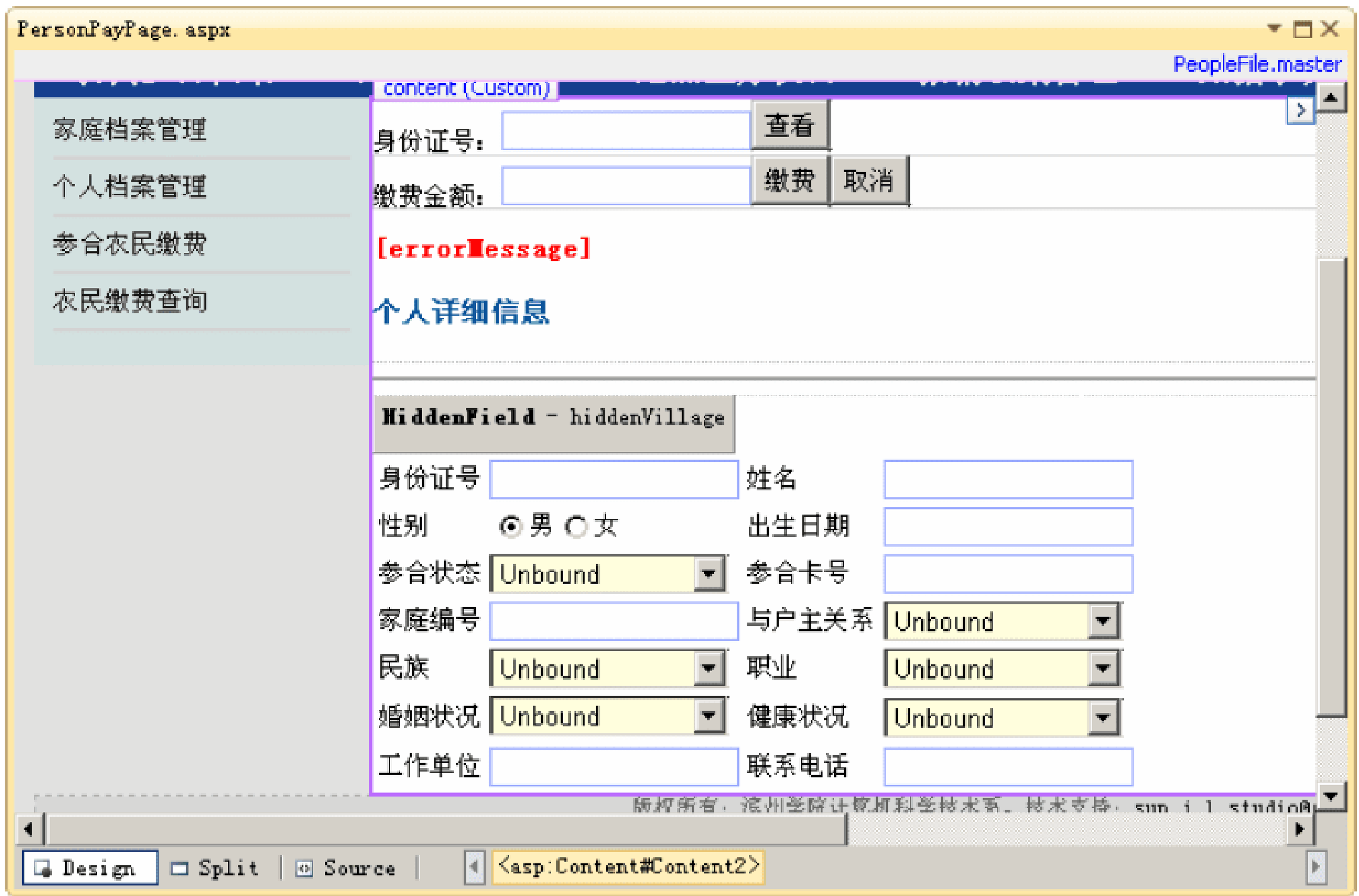


图 14.14　参合缴费页面布局

```
<asp:Content ID="Content2" ContentPlaceHolderID="content" runat="server">
    身份证号: <asp:TextBox runat="server" ID="personID" />
     <asp:Button runat="server" ID="ok" Text="查看" onclick="ok Click" />
    <asp:Panel ID="payPanel" Visible="false" runat="server">
    缴费金额: <asp:TextBox ID="money" runat="server" ReadOnly=
    "true"></asp:TextBox>
    <asp:Button ID="payButton" runat="server" Text="缴费" onclick=
    "Button1 Click" />
           <asp:Button ID="cancel" runat="server" Text="取消" onclick=
           "cancel Click"  />
     </asp:Panel>
    <br />
    <asp:Label ID="errorMessage" runat="server" Text="" CssClass=
    "error"></asp:Label> <br />
    <h3>个人详细信息</h3>   <hr />
    <uc1:PersonControl ID="PersonControl1" runat="server"/>
</asp:Content>
```

（4）在 PersonPayPage.aspx 页面的“查看”按钮的 Click 事件中，根据输入的个人编写显示个人详情。

```
protected void ok Click(object sender, EventArgs e)
{
    //根据输入的个人编号得到人员信息
    PersonInfo person = PersonBLL.getByID(personID.Text);
    if (person == null)
    {
        errorMessage.Visible = true;
        payPanel.Visible = false;
        errorMessage.Text = "未能找到此人信息。";
        return;
    }
    //将人员信息显示在页面底部的 PersonControl 控件中
    PersonControl1.person = person;
    int n = CompensateSpecialPolicyBLL.getPersonalPayment();
                                        //得到年度缴费金额
    money.Text = n.ToString();
```

```
    payPanel.Visible = true;
}
```

（5）在 PersonPayPage.aspx 页面的“缴费”按钮的 Click 事件中完成缴费。

```
protected void Button1_Click(object sender, EventArgs e)
{
    PersonPaymentBLL.pay(personID.Text);           //调用缴费方法
    clearInput();                                  //清除用户输入
    //向客户端注册 JavaScript 显示提示信息
    string script = "<script>alert('缴费成功! ');</script>";
    ClientScript.RegisterStartupScript(this.GetType(),        "success",
script);
}
```

（6）在表现层项目 Nrcm.Web 中添加一个页面 PayQueryPage.aspx 以实现缴费查询功能，页面布局如图 14.15 所示。

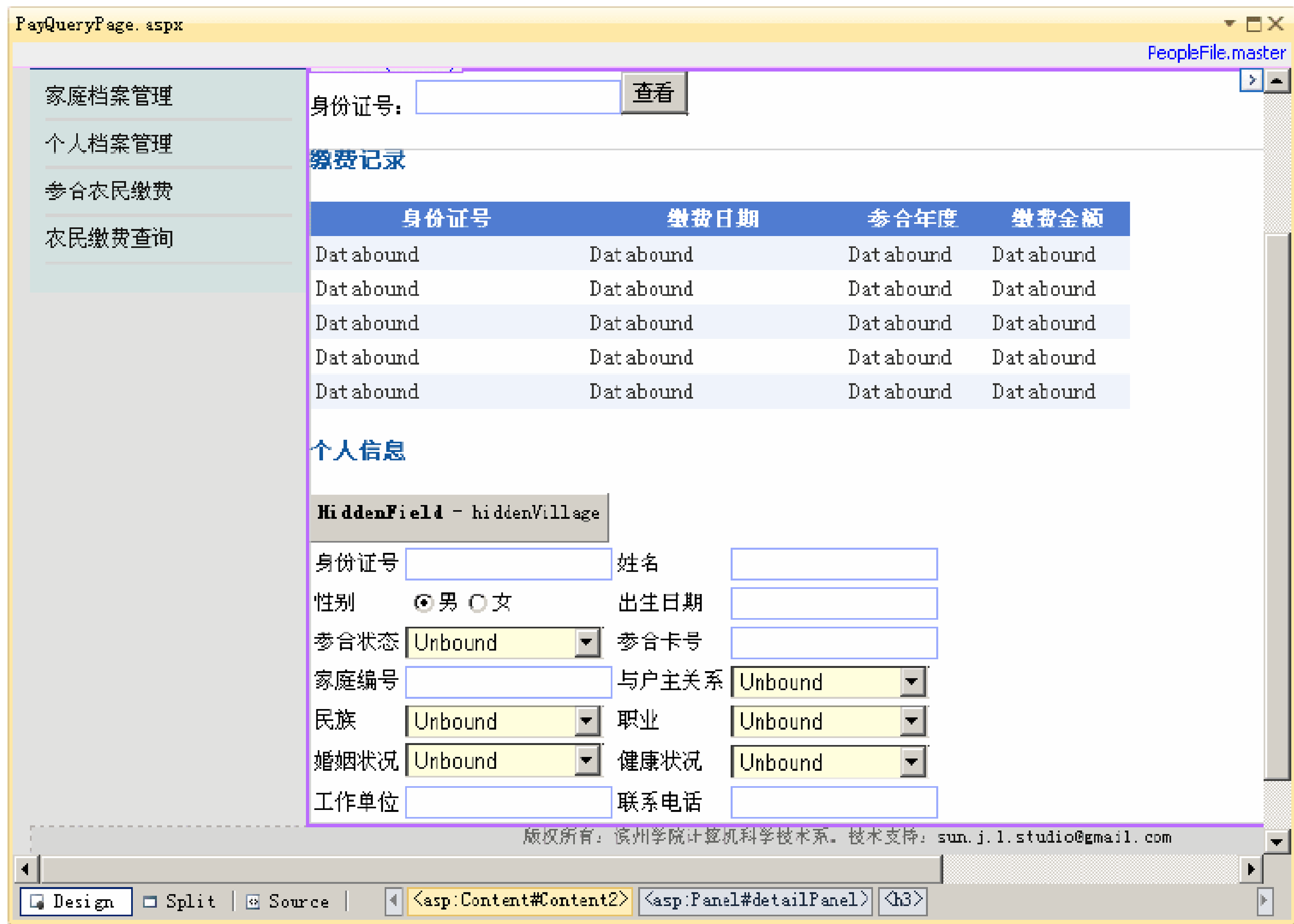


图 14.15　缴费查询页面

（7）在 PayQueryPage.aspx 页面的“查看”按钮的 Click 事件中，根据用户所输入的个人编号查询个人信息和缴费历史记录并显示在页面上。

```
protected void ok_Click(object sender, EventArgs e)
{
    PersonInfo person = PersonBLL.getByID(personID.Text);
                                          //根据 ID 得到个人信息
    //如果此人不存在，则隐藏明细信息，方法返回
    if (person == null)
    {
        detailPanel.Visible=false;
        return;
    }
    //如果此人存在，则显示详细信息，并显示缴费记录
    detailPanel.Visible=true;
    personDetail.person = person;
    var list = PersonPaymentBLL.getByPerson(personID.Text);
    grid1.DataSource=list;
```

```
    grid1.DataBind();
}
```

14.5　住院费用结算和审核

参合农民在定点医疗机构出院结算时，新农合系统需要计算可报销的金额，并将其从总费用中扣除，农民只需要缴纳剩余部分的费用。定点医疗机构每隔一段时间去新农合部门进行结算，集中领取农民住院费用中新农合报销部分的资金。

14.5.1　数据库表结构

住院费用结算是新农合系统中最为复杂的一个功能模块，涉及多个数据库表和复杂的计算规则。与住院费用结算相关的表主要有以下几个。

1. 费用项目表

该表保存了医院的收费项目（包括药品、化验检查、手术、材料费等）详细信息。表结构如下：

```
CREATE TABLE [dbo].[FYXM](
    [XM_CODE] [nvarchar](15)  NOT NULL,                    --项目编码
    [XM_NAME] [nvarchar](15)  NOT NULL,                    --项目名称
    [JSM] [nvarchar](10)  NULL,                            --拼音检索码
    [PL_CODE] [nchar](2)  NOT NULL,                        --品类代码
    [DJ] [float] NOT NULL,                                 --单价
    [BXBL] [float] NOT NULL DEFAULT ((0)),                 --报销比例
    [BXJB] [nchar](2)  NOT NULL  DEFAULT ((0)),            --报销级别
    [YPGN_CODE] [nchar](2)  NULL,                          --药品功能代码
    [GG] [nvarchar](10)  NULL,                             --规格
    [JX] [nvarchar](10)  NULL,                             --剂型
    [DW] [nvarchar](10)  NULL,                             --药品单位
    [BZDW] [nvarchar](10)  NULL,                           --包装单位
    [BZSL] [int] NULL DEFAULT ((0)),                       --包装数量
 CONSTRAINT [pk yyfyxm] PRIMARY KEY NONCLUSTERED
(   [XM_CODE] ASC   )
```

2. 住院费用表

该表保存了医院病人的消费明细数据。表结构如下：

```
CREATE TABLE [dbo].[PatientExpense](
    [RecordID] [bigint] IDENTITY(1,1) NOT NULL,      --流水号
    [YYBM] [nvarchar](10)  NOT NULL,                 --医院编码
    [ZYH] [nvarchar](10)  NOT NULL,                  --住院号
    [FYID] [nvarchar](20)  NULL,                     --费用 ID(HIS 中的主键)
    [RQ] [datetime] NULL,                            --费用日期
    [XM_CODE] [nvarchar](15)  NOT NULL,              --HIS 药品编码
    [PL_CODE] [nchar](3)  NULL,                      --品类编码
```

```
    [DJ] [money] NOT NULL  DEFAULT ((0)),           --单价
    [SL] [int] NOT NULL  DEFAULT ((0)),             --数量
    [ZJE] [money] NOT NULL  DEFAULT ((0)),          --金额
    [MZZYBZ] [nchar](1) NOT NULL,                   --门诊住院标志
    [TPBZ] [nchar](1) NULL DEFAULT ('0'),           --退费标志
    [CZY CODE] [nvarchar](10) NULL,                 --操作员代码
    [SHBZ] [nchar](2) NOT NULL  DEFAULT ('00'),     --审核标志
    [BXJE] [money] NOT NULL  DEFAULT ((0)),         --报销金额
    [DYXM] [nvarchar](15) NULL,                     --对应于新农合的药品编码
    [XM_NAME] [nvarchar](20) NULL,                  --药品名称
    [TranID] [bigint] NULL,                         --结算业务编号
    [YPJB] [nchar](2) NULL,                         --药品级别
    [YPXJ] [money] NULL,                            --药品限价
    [BXBL] [float] NULL,                            --报销比例
    [YPSH] [bit] NULL,                              --是否审核
 CONSTRAINT [PK PatientExpense  33D4B598] PRIMARY KEY CLUSTERED
(   [RecordID] ASC  )
```

3．住院病人表

该表保存了病人住院的相关信息，表结构如下：

```
CREATE TABLE [dbo].[HospitalPatient](
    [YYBM] [nvarchar](10) NOT NULL,                 --医院编码
    [ZYH] [nvarchar](10) NOT NULL,                  --住院号
    [RYRQ] [datetime] NULL,                         --入院日期
    [CYRQ] [datetime] NULL,                         --出院日期
    [KS CODE] [nchar](2) NULL,                      --科室代码
    [SFZH] [nvarchar](18) NOT NULL,                 --身份证号
    [XM] [nvarchar](10) NOT NULL,                   --姓名
    [XB] [nchar](1) NULL,                           --性别
    [CSRQ] [datetime] NULL,                         --出生日期
    [ZY] [nvarchar](10) NULL,                       --职业
    [DZ] [nvarchar](24) NULL,                       --地址
    [DH] [nvarchar](20) NULL,                       --电话
    [CW CODE] [nvarchar](10) NULL,                  --床位号
    [MZZYBZ] [nchar](1) NULL ,                      --门诊住院标志
 CONSTRAINT [PK Patient] PRIMARY KEY CLUSTERED
(   [YYBM] ASC, [ZYH] ASC   )
```

4．报销结算业务表

该表保存了住院病人结算汇总数据，每当有一个新农合病人出院结算时，就会产生一条汇总数据保存到此表。表结构如下：

```
CREATE TABLE [dbo].[CompensationTransaction](
    [RecordID] [bigint] IDENTITY(1,1) NOT NULL,             --流水号
    [TranDate] [datetime] NOT NULL DEFAULT (getdate()),     --业务发生日期
    [SFZH] [nvarchar](20) NOT NULL,                         --身份证号
    [YYBM] [nvarchar](10) NOT NULL,                         --医院编码
    [ZYH] [nvarchar](10) NOT NULL,                          --住院号
    [MZZYBZ] [char](1) NOT NULL,                            --门诊住院标志
    [FYZE] [money] NOT NULL DEFAULT ((0)),                  --费用总额
```

```
    [KBJE] [money] NULL,                                    --可报金额
    [SBJE] [money] NULL,                                    --实报金额
    [SHJE] [money] NULL,                                    --审核金额
    [SHBZ] [bit] NOT NULL  DEFAULT ((0)),                   --审核标志
    [SHYH] [nvarchar](10) NULL,                             --审核用户
    [SHRQ] [datetime] NULL,                                 --审核日期
    [JSBZ] [bit] NOT NULL DEFAULT ((0)),                    --结算标志
    [JSID] [int] NULL,                                      --结算编号
 CONSTRAINT [PK_CompensationTransaction] PRIMARY KEY CLUSTERED
(   [RecordID] ASC  )
```

5. 医疗机构表

该表保存了各个定点医疗机构的信息，表结构如下：

```
CREATE TABLE [dbo].[Hospital](
    [ID] [nvarchar](10) NOT NULL,                           --医院编码
    [Name] [nvarchar](30) NOT NULL,                         --医院名称
    [HospitalLevel] [nchar](2) NOT NULL,                    --医院级别
    [HospitalType] [nchar](2) NOT NULL,                     --医院类别
    [Address] [nvarchar](50) NULL,                          --地址
    [Post] [nvarchar](10) NULL,                             --邮编
    [Phone] [nvarchar](15) NULL,                            --联系电话
    [CEO] [nvarchar](10) NULL,                              --法人代表
    [Fund] [int] NULL,                                      --注册资金
    [PersonnelNum] [int] NULL,                              --联系人
    [BedNum] [int] NULL,                                    --床位数
 CONSTRAINT [PK_Hospital] PRIMARY KEY CLUSTERED
(
    [ID] ASC
)
```

14.5.2　住院费用结算

各个定点医疗机构一般都有医院管理系统（Hospital Information System，简称 HIS），病人住院、收费、出院等操作都通过医院的 HIS 系统进行管理。在医院就诊的病人有参合农民，也有其他病人。如果病人不是参合农民，那么病人从入院到出院的所有数据都与新农合系统无关。如果病人是参合农民，那么病人出院结算时，需要将病人住院费用导入新农合数据库来保存，并计算报销金额。住院费用报销金额的计算比较规则复杂，以下为计算步骤。

（1）得到病人的一条住院费用数据。此数据包含了使用什么药品或者做了什么检查项目（为了简单起见，下文中将药品和检查项目统称为药品），单价、数量、金额分别是多少。

（2）判断这种药品是否属于可报销药品。新农合系统有一个基本用药目录，只有在目录中的药品才是可报销药品。如果当前药品不是可报药品，则报销金额为 0，结束计算过程。

（3）判断这种药品报销级别和医院级别是否匹配。每种可报销药品都有一个报销级别，例如，某种药品报销级别为县级医院，则说明只有县级以上医院才能使用这种药品，如果在乡镇医院或者村卫生室使用此种药品则不予报销。如果一个药品是可报药品，需要检测

这个药品的报销级别和病人所住医院的级别。如果药品报销级别高于医院级别，则报销金额为 0，结束计算过程。

（4）检查药品限价。每种可报药品都有一个最高限价。如果病人住院时使用此药品的价格高于限价，则按照限价计算报销金额，否则按照实际价格计算报销金额。

（5）得到药品的报销比例。每种药品都可以设置一个报销比例，是全额还是按照一定比例报销。将第 4 步得到的药品价格乘以药品数量得到了药品金额，再乘以报销比例得到报销金额。

（6）将第 5 步得到的报销金额累加到总报销金额中。

（7）得到病人下一条住院费用数据，转第（2）步。如果已经处理完所有住院费用，则转第（8）步。

（8）根据分段报销比例和总报销金额，得到实际报销金额。

（9）检查此人当年度可报余额，如果可报余额不足，则实际报销金额为当年度可报余额。新农合系统规定，每人每年有个报销上限。报销上限减去当年度已经报销的金额即为当年度可报余额。

以上计算报销金额的过程涉及许多数据库操作，为了提高性能，在新农合系统中用存储过程来实现，存储过程代码如下：

```
-- =============================================
-- Description:计算报销费用
-- 找到对应药品，根据医院等级、药品限价和报销比例，计算药品可报金额
-- =============================================
ALTER PROCEDURE [dbo].[CalculateExpenseCompensation]
    @yybm nvarchar(10),                              --医院编码
    @zyh nvarchar(10),                               --住院号
    @mzzybz nchar(1)                                 --门诊住院标志：门诊住院
AS
BEGIN
set nocount on;
--首先检查相关参数：医院、病人、费用
execute CheckHospitalPatient @yybm,@zyh,@mzzybz
if @@error>0 return
declare @yyjb nchar(2);
select @yyjb=HospitalLevel from Hospital where ID=@yybm;
--此游标针对病人费用逐行读取，并修改对应项目、药品级别、药品限价、报销比例、报销金额
declare mycursor cursor local forward_only
for select recordid,xm_code,dj,sl,zje from PatientExpense
where YYBM=@yybm and ZYH=@zyh AND MZZYBZ=@mzzybz
for update of BXJE,DYXM,XM_NAME,YPJB,YPXJ,BXBL,YPSH;
declare @recordid bigint,@yyxm nvarchar(15);       --记录编号、医院编码
declare @dyxm nvarchar(15),@xmmc nvarchar(20);     --对应项目、项目名称
declare @dj money,@sl int ,@zje money;             --单价、数量、金额
declare @price_limit money;                        --限价
declare @bxbl float,@bxjb nchar(2);                --报销比例，报销级别
declare @shbz nchar(1)                             --审核标志
open mycursor;                                     --打开游标并循环读取数据
fetch next from mycursor into @recordid,@yyxm,@dj,@sl,@zje;
while @@fetch_status=0
begin
    update PatientExpense set dyxm=null,bxje=0,ypjb=null,ypxj=0,bxbl=0
    where current of mycursor;                     --为当前行设置初值
```

```
    --找到对应的药品（项目）
    set @dyxm=''
    set @shbz='0'
    select @dyxm=isnull(dyxm,''),@shbz=audit from MedicineMapping
    where YYBM=@yybm and YYXM=@yyxm;
    --未找到对应项目则继续下条费用
    if @dyxm=''  goto nextRow;
    if not exists
    (select * from fyxm where xm_code=@dyxm)
    goto nextRow;
    select @price_limit = isnull(dj,0), @bxbl = isnull(bxbl,0), @bxjb=
    isnull(bxjb,'01'), @xmmc = xm_name
    from fyxm where xm_code=@dyxm;                    //得到药品限价
    declare @ttt bit
    if @shbz='1'
        set @ttt=1
    else
        set @ttt=0
    UPDATE PatientExpense
    set DYXM = @dyxm, xm_name = @xmmc, ypjb = @bxjb, ypxj = @price_limit,
    bxbl = @bxbl,ypsh = @ttt
    where current of mycursor;
    if @bxjb>@yyjb goto nextRow;                  --药品级别高于当前医院级别
    if @shbz<>'1' goto nextrow;                   --未审核
    if @sl=0  begin set @dj=@zje; set @sl=1; end   --未输入数量则默认为 1
    if @dj>@price_limit  set @dj=@price_limit;
    update PatientExpense set BXJE=@dj*@sl*@bxbl where current of mycursor;
    nextRow:
    fetch next from mycursor into @recordid,@yyxm,@dj,@sl,@zje;
                                                  --取下条费用数据
end
END
```

14.5.3　住院业务审核

参合病人在定点医疗机构出院结算后，病人的住院信息和费用数据就被导入到新农合数据库中。为了防止医院做假，新农合管理部门需要对这些数据进行审核，然后以审核通过的数据为准拨付资金给定点医疗机构。审核不通过的数据将不能获得新农合报销资金。住院业务审核涉及两个页面：一个汇总页面和一个明细页面，下面将说明这两个页面及相关数据访问层、业务逻辑层的代码。

（1）在数据访问层项目 Nrcm.Dal 中添加一个类 CompensationTransactionDAL，实现与住院结算业务相关的数据访问操作，代码如下：

```
public static class CompensationTransactionDAL
{
    //根据业务编号得到业务详情
    public static CompensationTransaction getById(long id)
    {
        return EntityUtility.selectOne(getQuery(),t=>t.RecordID==id);
    }
    //得到指定医院指定日期范围内的业务列表
    public static List<CompensationTransaction> getByHospitalAndDate
    (string hospital, DateRange date,PageDataArgument page)
    {
```

```
        ObjectQueryArgument<CompensationTransaction,long> arg=new
        ObjectQueryArgument<CompensationTransaction,long>(getQuery(),
        t=>t.RecordID,page);
        return EntityUtility.selectMany<CompensationTransaction, long>
        (arg, t => t.YYBM == hospital && t.TranDate >= date.from && t.TranDate
        <= date.to);
    }
    //添加一个业务
    public static int add(CompensationTransaction tran)
    {
        return EntityUtility.add(new ExpenseContext(), tran);
    }
    private static ObjectQuery<CompensationTransaction> getQuery()
    {
        return new ExpenseContext().CompensationTransaction;
    }

}
```

（2）在业务逻辑层项目 Nrcm.Bll 中添加一个类 CompensationTransactionBLL。

（3）在表现层项目 Nrcm.Web 中添加一个页面 TransactionAuditPage.aspx，页面布局如图 14.16 所示。



图 14.16　住院业务审核页面布局

在图 14.16 所示的 TransactionAuditPage.aspx 页面中，在页面顶部选择一个定点医疗机构，输入一个日期范围，单击“查看”按钮，即可查看符合条件的住院结算业务。单击数据列表中的“费用明细”图片按钮，即可打开另外一个页面，查看此次业务的费用明细。页面代码如下：

```
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <%--导入 JavaScript 文件和 CSS 文件--%>
    <link href="../css/ui-lightness/jquery-ui-1.7.2.css" rel="stylesheet"
    type="text/css" />
    <script type="text/javascript" src="../js/jquery-1.3.2.js"></script>
    <script src="../js/jquery-ui-1.7.2.js" type="text/
    javascript"></script>
    <script src="../js/MyUtility.js" type="text/javascript"></script>
    <script type="text/javascript">
        $(function () {
            //为 GridView 添加光棒效果
```

```
            $('table.grid').find('tr').hover(
            function () { $(this).addClass("hoverRow"); },
            function () { $(this).removeClass("hoverRow"); });
            $SjlUtility.addButtonClass();
            //使用 jQuery 日历扩展两个日期文本框
            $SjlUtility.jQueryDatePickerChinese();
            $('#<%=date1.ClientID%>').datepicker();
            $('#<%=date2.ClientID%>').datepicker();
        });         //$(function)
    </script>
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="content" runat="server">
    <h3>住院业务审核</h3>
    医院：<asp:DropDownList runat="server" ID="hospitalList" DataTextField=
    "Name" DataValueField="ID"></asp:DropDownList>
    日期范围：<asp:TextBox runat="server" ID="date1" />
    至 <asp:TextBox runat="server" ID="date2" />
    <asp:Button Text="查看" runat="server" ID="ok" onclick="ok Click" />
    <%--住院结算业务列表--%>
    <asp:GridView ID="grid" runat="server" AutoGenerateColumns="False"
    DataKeyNames="RecordID" CssClass="grid" >
        <Columns>
            <asp:BoundField HeaderText="流水号" DataField="RecordID"
            Visible="false"  />
            <asp:BoundField HeaderText="结算日期" DataField="TranDate"
            DataFormatString="{0:d}" />
            <asp:BoundField HeaderText="住院号" DataField="ZYH"  />
            <asp:BoundField HeaderText="身份证号" DataField="SFZH"  />
            <asp:BoundField HeaderText="费用总额" DataField="FYZE" />
            <asp:BoundField HeaderText="可报金额" DataField="KBJE"  />
            <asp:BoundField HeaderText="实报金额" DataField="SBJE" />
            <asp:CheckBoxField HeaderText="审核标志" DataField="SHBZ"
            ReadOnly="true" />
            <asp:BoundField HeaderText="审核金额" DataField="SHJE"  />
            <asp:CheckBoxField HeaderText="结算标志" DataField="JSBZ"
            ReadOnly="true"  />
            <asp:BoundField HeaderText="结算编号" DataField="JSID" />
             <%--以下列为模板列，包含一个图片链接，链接到另一个页面显示费用明细--%>
            <asp:TemplateField HeaderText="费用明细">
             <ItemTemplate>
             <a href='ExpenseDetialPage.aspx?tran=<%#Eval("RecordID") %>'
             target=" blank">
             <img  style="border:none; width:20px;" src="../images/search.
             png" alt="费用明细"/></a>
             </ItemTemplate>
             </asp:TemplateField>
        </Columns>
        <RowStyle HorizontalAlign="Center"  />
    </asp:GridView>
    <webdiyer:AspNetPager ID="pager1" runat="server" onpagechanged=
    "pager1 PageChanged">
    </webdiyer:AspNetPager>
</asp:Content>
```

（4）在 TransactionAuditPage.aspx 页面的 Page_Load 事件中，绑定医院列表。

```
protected void Page Load(object sender, EventArgs e)
{
```

```
    if (!IsPostBack)
        bindHospitals();
}
//绑定医院列表
private void bindHospitals()
{
    var list = HospitalBLL.getAll(PageDataArgument.allData);
    hospitalList.DataSource = list;
    hospitalList.DataBind();
}
```

（5）在 TransactionAuditPage.aspx 页面上“查看”按钮的 Click 事件中，根据用户输入的查询条件获取数据并显示在页面上。

```
protected void ok Click(object sender, EventArgs e)
{
    pager1.CurrentPageIndex = 1;
    bindList(true);
}
private void bindList(bool refreshCount)
{
    string hospital = hospitalList.SelectedValue;      //得到所选中的医院
    DateRange date = DateRange.between2Date(date1.Text, date2.Text);
                                                       //获得日期范围
    PageDataArgument page = new PageDataArgument();
    page.pageSize = pager1.PageSize;
    page.refreshCount = refreshCount;
    page.pageIndex = pager1.CurrentPageIndex-1;
    //查询数据并绑定到GridView控件
    var list = CompensationTransactionBLL.getByHospitalAndDate
    (hospital,date,page);
    grid.DataSource = list;
    grid.DataBind();
}
protected void pager1 PageChanged(object sender, EventArgs e)
{
    bindList(false);
}
```

（6）运行 TransactionAuditPage.aspx 页面，运行界面如图 14.17 所示。



图 14.17　住院业务查询审核页面

（7）在数据访问层项目 Nrcm.Dal 中添加一个 PatientExpenseDAL 类，实现与病人费用相关的数据访问功能。

```
public static class PatientExpenseDAL
{
    /// <summary>
    /// 得到某次结算业务所包含的费用数据
    /// </summary>
    /// <param name="tran">结算业务编号</param>
    /// <param name="page">分页参数</param>
    public static List<PatientExpense> getByTransaction(long tran,PageData
    Argument page)
    {
        var query = getQuery();
        ObjectQueryArgument<PatientExpense, string> arg
            = new ObjectQueryArgument<PatientExpense, string>();
        arg.query = query;
        arg.page = page;
        arg.sort = e=>e.FYID;
        var list=EntityUtility.selectMany(arg,e => e.TranID == tran);
        list.ForEach(e => loadDetail(e));
        return list;
    }
    /// <summary>
    /// 审核某个明细费用
    /// </summary>
    /// <param name="recordId">费用记录编号</param>
    /// <param name="state">审核状态</param>
    public static void audit(long recordId,AuditStateEnum state)
    {
        string val=null;
        if (state == AuditStateEnum.None)
            val = "00";
        else if (state == AuditStateEnum.Approve)
            val = "01";
        else if (state == AuditStateEnum.Deny)
            val = "02";
        else
            throw new ApplicationException("指定的审核状态不合法");
        var query = getQuery();
        var item = (from ex in query
                    where ex.RecordID == recordId
                    select ex).FirstOrDefault();
        if (item == null)
            throw new ApplicationException("未找到费用数据") ;
        item.SHBZ = val;
        query.Context.SaveChanges();
    }
    /// <summary>
    /// 得到一次结算的审核汇总数据并保存到数据库
    /// </summary>
    /// <param name="tranId"></param>
    /// <param name="user"></param>
    public static void auditSummary(long tranId,string user)
    {
        using (ExpenseContext context = new ExpenseContext())
        {
            Database db = DatabaseFactory.CreateDatabase("NRCM");
            db.ExecuteNonQuery("ShenHe", tranId, user);      //调用存储过程
```

```
ShenHe 实现
            }
        }
        //加载费用数据的外键信息
        private static void loadDetail(PatientExpense expense)
        {
            var item=FyxmDAL.getById(expense.XM_CODE);
            expense.xmmc = item == null ? "" : item.XM_NAME;    //药品名称
            var audit = SmallDictionaryDAL.getByID(SmallDictionaryTables.
            TableAuditState, expense.SHBZ);
            expense.audit = audit == null ? "" : audit.name;   //审核状态
        }
        private static ObjectQuery<PatientExpense> getQuery()
        {
            return new ExpenseContext().PatientExpense;
        }
}
```

上述代码的 auditSummary()方法用到的存储过程 ShenHe 代码如下：

```
CREATE PROCEDURE [dbo].[ShenHe]
--审核一笔业务并产生相应数据
@tran bigint,                                          --被审核的业务 ID
@user nvarchar(10)                                     --审核人
AS
BEGIN
    set nocount on;
    declare @je money
    set @je=0;
    --得到所有已经通过审核的费用总额
    select @je=sum(isnull(bxje,0)) from PatientExpense
    where tranid=@tran and SHBZ='01';
    --根据分段报销比例计算应得报销金额
    if @je is not null
        select @je=dbo.CalculateCompensation(@je);
    --保存数据
    update CompensationTransaction
    set SHYH=@user,SHRQ=getdate(),SHBZ=1,SHJE=@je
    where RecordID=@tran;
END
```

（8）在表现层项目 Nrcm.Web 中添加一个费用明细页面 ExpenseDetialPage.aspx，此页面可以显示某次住院结算所包含的所有费用明细，页面布局如图 14.18 所示。



图 14.18　费用明细页面布局

在图 14.18 所示的 ExpenseDetialPage.aspx 页面中，根据 QueryString 中传递的业务编号查找此业务所包含的所有费用数据并显示在 GridView 中。用户可以单击某条费用数据后面的“通过”或“驳回”按钮实现对单条费用的审核功能，也可以单击“全部通过审核”按钮全部审核当前页面显示的所有费用数据。用户单击“保存审核数据”按钮时，将对此次审核数据进行汇总，并保存到数据库。ExpenseDetialPage.aspx 页面代码如下：

```
<form id="form1" runat="server">
<h3>住院费用明细</h3>
<div>
    页面大小：<asp:TextBox runat="server" ID="pageSize" Text="10" />
    <asp:Button runat="server" ID="ok" Text="改变页面大小" onclick=
    "ok_Click" />
    <asp:Button runat="server" ID="approveAll" Text="全部通过审核"
        onclick="approveAll_Click" />
    <asp:Button runat="server" ID="audit" Text="保存审核数据" onclick=
    "audit_Click" /><br />
    <asp:GridView ID="grid" runat="server" CssClass="grid"
    CellPadding="8"
        AutoGenerateColumns="False" DataKeyNames="RecordID"
        onrowcommand="grid_RowCommand">
        <Columns>
            <asp:BoundField DataField="RecordID" HeaderText="RecordID"
            ReadOnly="True" Visible="false"/>
            <asp:BoundField DataField="RQ" HeaderText="日期"
            DataFormatString="{0:g}"/>
            <asp:BoundField DataField="XM_NAME" HeaderText="药品项目"   />
            <asp:BoundField DataField="DJ" HeaderText="单价" />
            <asp:BoundField DataField="YPXJ" HeaderText="限价"/>
            <asp:BoundField DataField="SL" HeaderText="数量"  />
            <asp:BoundField DataField="ZJE" HeaderText="金额" />
            <asp:BoundField DataField="audit" HeaderText="审核状态" />
            <asp:BoundField DataField="BXJE" HeaderText="报销金额" />
            <asp:BoundField DataField="BXBL" HeaderText="报销比例" />
            <%--以下模板列中包含两个命令按钮，一个"通过"和一个"驳回"--%>
            <asp:TemplateField HeaderText="审核">
            <ItemTemplate>
            <asp:LinkButton ID="approve" runat="server" CommandName=
            "approve" CommandArgument='<%#Eval("RecordID")%>' Text="通过"
            />
            <asp:LinkButton ID="deny" runat="server" CommandName="deny"
            Text="驳回" CommandArgument='<%#Eval("RecordID")%>' />
            </ItemTemplate>
            </asp:TemplateField>
        </Columns>
    </asp:GridView>
    <webdiyer:AspNetPager ID="pager1" runat="server"
        onpagechanged="pager1_PageChanged">
    </webdiyer:AspNetPager>
</div>
</form>
```

（9）在 ExpenseDetialPage.aspx 页面的 Page_Load 事件中，查询费用明细数据并显示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
```

```
    {
        bindList(true);
    }
}
/// <summary>
/// 绑定费用明细数据列表
/// </summary>
/// <param name="firstTime">是否第一次加载数据，如果是则需要刷新总记录数</param>
private void bindList(bool firstTime)
{
    string tran = Request.QueryString["tran"];          //得到业务编号
    long id;
    if (!long.TryParse(tran, out id))
        return;
    //根据分页控件得到分页参数
    PageDataArgument page = new PageDataArgument();
    page.pageSize = pager1.PageSize;
    page.refreshCount = true;
    page.pageIndex = pager1.CurrentPageIndex - 1;
    //执行查询并绑定数据
    List<PatientExpense> list = PatientExpenseBLL.getByTransaction(id,
    page);
    if(firstTime)
        pager1.RecordCount=page.count;
    grid.DataSource = list;
    grid.DataBind();
}
```

（10）在 ExpenseDetialPage.aspx 页面上“改变页面大小”按钮的 Click 事件和分页控件的 PageChanged 事件中，根据新的页面大小和页码绑定数据。

```
protected void pager1 PageChanged(object sender, EventArgs e)
{
    bindList(false);
}
protected void ok Click(object sender, EventArgs e)
{
    pager1.PageSize = int.Parse(pageSize.Text);
    bindList(false);
}
```

（11）在 ExpenseDetialPage.aspx 页面上 GridView 控件的 RowCommand 事件中，对当前费用进行审核操作（通过或驳回）。

```
protected void grid RowCommand(object sender, GridViewCommandEventArgs e)
{
    AuditStateEnum audit = AuditStateEnum.Invalid ;
    //根据 CommandName 设置审核状态
    if (e.CommandName == "approve")
        audit = AuditStateEnum.Approve;
    else if (e.CommandName == "deny")
        audit = AuditStateEnum.Deny;
    long id = Convert.ToInt64(e.CommandArgument);
    PatientExpenseBLL.audit(id, audit);
    bindList(false);                                              //重新绑定数据
}
public enum AuditStateEnum
{   None,   Approve,    Deny,   Invalid }
```

（12）在 ExpenseDetialPage.aspx 页面上“全部通过审核”按钮的 Click 事件中，将页面上显示的所有费用都设置为审核通过。

```
protected void approveAll Click(object sender, EventArgs e)
{
    for (int i = 0; i < grid.Rows.Count; i++)
    {
        long id = Convert.ToInt64(grid.DataKeys[i][0]);
        PatientExpenseBLL.audit(id, AuditStateEnum.Approve);
    }
    bindList(false);
}     protected void approveAll Click(object sender, EventArgs e)
{
    //对 GridView 的所有行进行循环，设置每行数据的审核状态为通过
    for (int i = 0; i < grid.Rows.Count; i++)
    {
        long id = Convert.ToInt64(grid.DataKeys[i][0]);
        PatientExpenseBLL.audit(id, AuditStateEnum.Approve);
    }
    bindList(false);                                    //重新绑定数据
}
```

（13）在 ExpenseDetialPage.aspx 页面上“保存审核数据”按钮的 Click 事件中，将审核汇总数据保存到数据库。

```
protected void audit Click(object sender, EventArgs e)
{
    string tran = Request.QueryString["tran"];
    long id;
    if (!long.TryParse(tran, out id))
        return;
    PatientExpenseBLL.auditSummary(id, WebUtility.currentUser.ID);
}
```

（14）在浏览器中查看 ExpenseDetialPage.aspx 页面，运行界面如图 14.19 所示。



图 14.19　费用明细审核页面

14.6 小　　结

本章介绍了新型农村合作医疗管理信息系统的设计与实现。新农合系统业务较为复杂，功能模块较多，受篇幅限制，本章仅介绍了几个典型的功能模块，此项目完整代码可参照配套光盘。读者可通过光盘中的源代码，了解更多的开发技巧。

附录　Visual Studio 操作快捷键

本附录主要是根据一些开发人员的工作经验而总结的，目的就是增加开发速度和效率。下面是一些常用快捷键，列表如下所示。

快　捷　键	功　　能
F12	转到定义
F1	帮助
F5	运行程序并调试
F10	跨过程序执行
F7	切换到代码或者 ASPX
F4	查看属性
F11	逐语句调试
F10	逐过程调试
Ctrl +KCtrl+D	格式化文档
Ctrl +KCtrl+C	注释代码段
Ctrl +KCtrl+U	取消注释
Ctrl +KCtrl+L	取消标签
Ctrl +MCtrl+M	折叠代码段
Alt+F4	关闭 Visual Studio 开发工具界面
Ctrl +F4	关闭单独页面
Shift+F5	停止调试
Ctrl+J	提示名称
Alt+→	提示并选择最接近的名称
Ctrl +L	删除选定行
Ctrl + Shift +空格	出现参数列表提示
Ctrl＋R＋E	把字段封装成属性
Ctrl＋Shift＋B	编译程序
Ctrl＋左右方向键	按单词移动光标，准确快速
Ctrl +R＋R	重命名变量、方法等的名称
Ctrl+B	定义书签
Ctrl+N	转到下一个书签（方便修改代码）
Shift+F7	切换代码和页面
Alt+B+E	重新生成
Ctrl+F7	生成编译
Ctrl+O	打开文件
Ctrl+Shift+O	打开项目
Ctrl+Shift+C	显示类视图窗口

（续表）

快　捷　键	功　　能
Shift+F4	显示项目属性窗口
Ctrl+Shift+E	显示资源视图
Ctrl+F12	转到声明
Ctrl+Alt+J	对象浏览
Ctrl+Alt+F1	帮助目录
Ctrl+F1	动态帮助
Shift+F1	当前窗口帮助
Ctrl+Alt+F3	帮助——搜索
Shift+Alt+ENTER	全屏显示
Ctrl+-	向后定位
Ctrl+Shift+-	向前定位
Ctrl+F4	关闭文档窗口
Ctrl+Page Down	光标定位到窗口上方
Ctrl+Page Up	光标定位到窗口下方
Ctrl+Tab	下一个文档窗口
Ctrl+K，Ctrl+L	取消 remark
Ctrl+K，Ctrl+C	注释选择的代码
Ctrl+K，Ctrl+U	取消对选择代码的注释
Ctrl+M，Ctrl+O	折叠代码定义
Ctrl+M，Ctrl+L	展开代码定义
Ctrl+Delete	删除至词尾
Ctrl+Backspace	删除至词头
Shift+Tab	取消制表符
Ctrl+U	转小写
Ctrl+Shift+U	转大写
Ctrl+Shift+End	选择至文档末尾
Ctrl+Shift+Home	选择至文档末尾开始
Shift+End	选择至行尾
Shift+Home	选择至行开始处
Shift+Alt+End	垂直选择到最后尾
Shift+Alt+Home	垂直选择到最前面
Ctrl+A	全选
Ctrl+W	选择当前单词
Ctrl+Shift+Page Up	选择至本页前面
Ctrl+Shift+Page Down	选择至本页后面
Ctrl+End	文档定位到最后
Ctrl+Home	文档定位到最前
Ctrl+G	转到…
Ctrl+K，Ctrl+P	上一个标签
Ctrl+K，Ctrl+N	下一个标签
Alt+F10	调试—ApplyCodeChanges

（续表）

快　捷　键	功　　能
Ctrl+Alt+Break	停止调试
Ctrl+Shift+F9	取消所有断点
Ctrl+F9	允许中断
Ctrl+F5	运行不调试

下面分类介绍一些快捷键，某些可能与上面有重复，主要是增加读者印象。

（1）调试快捷键列表。

快　捷　键	功　　能
F6	生成解决方案
Ctrl+F6	生成当前项目
F7	查看代码
Shift+F7	查看窗体设计器
F5	启动调试
Ctrl+F5	开始执行（不调试）
Shift+F5	停止调试
Ctrl+Shift+F5	重启调试
F9	切换断点
Ctrl+F9	启用/停止断点
Ctrl+Shift+F9	删除全部断点
F10	逐过程
Ctrl+F10	运行到光标处
F11	逐语句

（2）编辑快捷键列表。

快　捷　键	功　　能
Shift+Alt+Enter	切换全屏编辑
Ctrl+B，T	切换书签开关
Ctrl+B，N	移动到下一书签
Ctrl+B，P	移动到上一书签
Ctrl+B，C	清除全部标签
Ctrl+I	渐进式搜索
Ctrl+Shift+I	反向渐进式搜索
Ctrl+F	查找
Ctrl+Shift+F	在文件中查找
F3	查找下一个
Shift+F3	查找上一个
Ctrl+H	替换
Ctrl+Shift+H	在文件中替换
Alt+F12	查找符号（列出所有查找结果）

（续表）

快　捷　键	功　　能
Ctrl+Shift+V	剪贴板循环
Ctrl+左右箭头键	一次可以移动一个单词
Ctrl+上下箭头键	滚动代码屏幕，但不移动光标位置
Ctrl+Shift+L	删除当前行
Ctrl+M，M	隐藏或展开当前嵌套的折叠状态
Ctrl+M，L	将所有过程设置为相同的隐藏或展开状态
Ctrl+M，P	停止大纲显示
Ctrl+E，S	查看空白
Ctrl+E，W	自动换行
Ctrl+G	转到指定行
Shift+Alt+箭头键	选择矩形文本
Alt+鼠标左按钮	选择矩形文本
Ctrl+Shift+U	全部变为大写
Ctrl+U	全部变为小写

（3）代码快捷键列表。

快　捷　键	功　　能
Ctrl+J	列出成员
Ctrl+Shift+空格键	参数信息
Ctrl+K，I	快速信息
Ctrl+E，C	注释选定内容
Ctrl+E，U	取消选定注释内容
Ctrl+K，M	生成方法存根
Ctrl+K，X	插入代码段
Ctrl+K，S	插入外侧代码

（4）有关窗口的快捷键列表。

快　捷　键	功　　能
Ctrl+W，W	浏览器窗口
Ctrl+W，S	解决方案管理器
Ctrl+W，C	类视图
Ctrl+W，E	错误列表
Ctrl+W，O	输出视图
Ctrl+W，P	属性窗口
Ctrl+W，T	任务列表
Ctrl+W，X	工具箱
Ctrl+W，B	书签窗口
Ctrl+W，U	文档大纲
Ctrl+D，B	断点窗口

（续表）

快　捷　键	功　　能
Ctrl+D，I	即时窗口
Ctrl+Tab	活动窗体切换
Ctrl+Shift+N	新建项目
Ctrl+Shift+O	打开项目
Ctrl+Shift+S	全部保存
Shift+Alt+C	新建类
Ctrl+Shift+A	新建项

（5）Visual Studio 中没有明确指出的快捷键。

快　捷　键	功　　能
Ctrl+Space	直接完成类或函数（本来这个并不算隐藏的快捷键，但是因为中文输入法抢占了这个快捷键，替代的快捷键是 Alt+Right）
Shift+Delete	整行删除，并且将这一行放到剪贴板（这时候不能选中一段内容）
Shift+Insert	粘贴，使用 Ctrl+V 的功能类似，主要是为了和 Shift+Delete 对应
Ctrl+Up，Ctrl+Down	滚动编辑器，但尽量不移动光标，光标保证在可见范围内
Ctrl+BackSpace，Ctrl+Delete	整词删除，有的时候很有用
Ctrl+Left，Ctrl+Right	按整词移动光标（不算隐藏，和前面几条加起来就是 Ctrl 光标控制套件了）
Alt+Shift+F10	打开执行改名，实现接口和抽象类的小窗口（还可以用 Ctrl+.，不过有的中文输入法用到这个）
Shift+F9	调试是打开 QuickWatch，内容是当前光标所在处的内容
Shift+F12	查找所有引用
Ctrl+F10=F5	开始 Debug
Ctrl+F6	循环查看代码窗口，有点 Ctrl+Tab 的感觉
Ctrl+F3	查找当前光标选中的内容，可以和 F3 配合使用
Ctrl+F2	将焦点转移到类的下拉框上
Alt+F7	Ctrl+Tab，下一个文档窗口
Alt+F11	新开 Visual Studio 并编辑宏
Alt+F12	查找，等同于 Ctrl+F